Vrije Universiteit Brussel

Faculteit van de Wetenschappen
Vakgroep Computerwetenschappen
Laboratorium voor Programmeerkunde

# Handling Partial Failures in Mobile Ad hoc Network Applications: From Programming Language Design to Tool Support

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Elisa González Boix

Promotoren: Prof. Dr. Wolfgang De Meuter en Prof. Dr. Theo D'Hondt

October 2012

Print: Silhouet, Maldegem

2012 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. ++32 (0)2 289 26 50
Fax ++32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 223 5
NUR 989
Legal deposit D/2012/11.161/159

*To my parents, Feliciano and Evelin,*
whose endless support made this dissertation possible.

*To my brother, Daniel,*
who inspired me to study computer science.

*To Christophe, my love,*
who inspires me to keep on pressing forward.

# Abstract

Progress in the field of wireless technology has resulted in a growing body of research that deals with *mobile ad hoc networks* (MANETs): networks composed of *mobile* devices that are connected by *wireless* communication links with a limited communication range. The limited communication range of devices combined with the fact that devices move about renders applications subject to higher rates of partial failures than in traditional, stationary networks. This dissertation investigates programming language support to deal with the effects engendered by *partial failures* in the software development process. In particular, we investigate support for handling partial failures in MANET applications, from the programming language design to tool support.

In order to support distributed programming in such a dynamically changing environment, it is worth to investigate software engineering techniques where network disconnections are not treated as the exception but the rule. This observation motivated the development of the *ambient-oriented programming paradigm* in which this work is rooted. Within the context of this paradigm, we propose expressive abstractions that aid developers to detect, reason about, and handle partial failures. Our survey of related work will reveal that *leasing* provides a solution for a failure handling model in MANETs. However, to date no leasing model has been designed specifically for MANETs. In this context, leasing needs to be reconciled with computational models that deal with transient failures, and provide a well-defined high-level interface to allow developers to handle failures in a modular and reusable way, while accommodating the various application needs. This work studies a failure handling model which combines leasing with ambient-oriented programming, leading to the concept of *ambient-oriented leasing*.

We investigate ambient-oriented leasing in the two distinct distributed computing models. First, we integrate our leasing model in a distributed object-oriented model as a novel referencing abstraction, called *leased object references*. Second, we explore the applicability of ambient-oriented leasing to tuple spaces, resulting in a novel adaptation of the tuple space model for MANETs called *TOTAM* (Tuples on the Ambient). We design and implement both incarnations of ambient-oriented leasing in *AmbientTalk/M*, an ambient-oriented language which provides a unified meta model on which both language abstractions as well as tool support is built.

To support the development of MANET applications, we study tool support in the form of an *ambient-oriented debugger* that helps programmers to achieve a better understanding of the dynamic behaviour of a MANET application. As a proof-of-concept tool we provide an online ambient-oriented debugger integrated into AmbientTalk Eclipse IDE called *REME-D* (Reflective Epidemic MEssage-oriented Debugger).

Together, the programming language abstractions and the debugging support presented in this dissertation provide an innovative toolbox that allows MANET application developers to deal with partial failures in an anticipated and controllable way.

# Samenvatting

Innovaties in het domein van draadloze technologie hebben geleid tot een onderzoeksstroming die zich toespitst op zogenaamde mobiele ad-hoc netwerken (MANETs): dit zijn netwerken waarbij *mobiele* apparaten met elkaar verbonden zijn door middel van *draadloze* communicatie-links die gelimiteerd zijn in hun reikwijdte. De gelimiteerde reikwijdte van de communicatie in combinatie met het feit dat deze toestellen mobiel, zijn leidt ertoe dat toepassingen die gebruik maken van zulke netwerken veel meer blootgesteld worden aan partiële fouten dan toepassingen die gebruik maken van een traditionele gedistribueerde netwerkinfrastructuur. Deze thesis onderzoekt programmeerabstracties die het mogelijk maken om om te springen met de nefaste effecten als gevolg van *partiële fouten* tijdens het software-ontwikkelingsproces. In het bijzonder onderzoeken we ondersteuning voor het omspringen met partiële fouten voor toepassingen die draaien op MANETs. Dit gaat van programmeertaalontwerp tot tool-ondersteuning.

Om ondersteuning te geven tijdens het programmeren van gedistribueerde applicaties in zulke dynamisch veranderende omgeving, is het nuttig om software engineering-technieken te onderzoeken waarbij het verbreken van een netwerkverbinding niet gezien wordt als een uitzondering maar eerder als de regel. Deze observatie heeft de ontwikkeling van het *ambient-georiënteerd paradigma* gemotiveerd waar ook dit werk zijn wortels vindt. In de context van dit paradigma stellen wij expressieve abstracties voor die de ontwikkelaar helpen om partiële fouten te detecteren, erover te redeneren, en ermee om te springen. Ons overzicht van gerelateerd werk zal onthullen dat *leasing* een oplossing biedt als basis voor een foutenafhandelingsmodel in MANETs. Echter, tot op heden is er geen leasing-model ontwikkeld specifiek voor MANETs. In deze context moet leasing verzoent worden met modellen die omspringen met tijdelijke fouten, en moet er een welgedefinieerde interface van een voldoende hoog abstratieniveau aangeboden worden. Dit moet de programmeur toelaten om te springen met fouten in een modulaire en herbruikbare manier, zonder de verschillende software-vereisten uit het oog te verliezen. Dit werk bestudeert een foutenafhandelingsmodel dat leasing verzoent met ambient-georiënteerd programmeren. Het resulterende model noemen we *ambient-oriented leasing*.

We onderzoeken ambient-oriented leasing in twee verschillende gedistribueerde programmeermodellen. Ten eerste, integereren we ons model met een gedistribueerd object-georiënteerde programmeermodel door middel van een innovatief referentiemechanisme genaamd *leased object references*. Ten tweede, exploreren we de toepasbaarheid van ambient-oriented leasing-abstracties voor tupelruimtes, wat resulteert in een vernieuwende adaptatie van het tupelruimtemodel voor MANETs genaamd *TOTAM* (Tuples on the Ambient). We ontwerpen en ontwikkelen beide incarnaties van ambient-oriented leasing in *AmbientTalk/M*, een ambient-georiënteerde taal die een geunificeerd metamodel aanbiedt waar zowel taalabstracties als tools op gebouwd kunnen

worden.

Om ondersteuning aan te bieden voor de ontwikkeling van toepassingen die op MANETs draaien bestuderen we tool-ondersteuning onder de vorm van een *ambient-oriented debugger* dewelke helpt om programmeurs een beter inzicht te geven in het dynamisch gedrag van deze toepassingen. We bieden een prototype aan van een online ambient-oriented debugger debugger die geïntegreerd is met de ontwikkelingsomgeving voor AmbientTalk in Eclipse genaamd *REME-D* (Reflective Epidemic MEssage-oriented Debugger).

Tesamen bieden de programmeerabstracties en de ondersteuning voor debugging een innovatie set van werktuigen die de programmeur in staat stelt om partiële fouten het hoofd te bieden in een geanticipeerde en controleerbare manier.

# Acknowledgements

This dissertation would have not been possible without the help and support of my colleagues, friends and family.

First of all, I would like to sincerely thank both of my promotors, Theo D'Hondt and Wolfgang De Meuter, for unconditionally supporting my work throughout all these years, and finding the means to fund me. I am greatly indebted to Theo who gave me the opportunity to become a researcher at his lab. I came to Brussels with a 4-month Erasmus grant to do my master thesis, and he succeeded in inspiring me to do research through his classes and the intellectually stimulating environment that this lab (still called PROG in those days) is. His thought-provoking classes spurred up my interest in programming language design. I am also deeply grateful to Wolf, discussions with whom have tremendously helped in crystallizing ideas, and improving the quality and clarity of my papers and, in particular, this dissertation. Despite being a busy man, he has always made time when it was necessary and never hesitated to talk via video conference or even to come over to my place. I also appreciate the freedom that he has always given me; in particular, in the context of our university's Distributed and Mobile Programming Paradigms course. Thanks Theo and Wolf!

I owe a big "thank you" to Carlos Noguera, who guided me through the writing process, proofread the whole text, and gave me all those useful comments, and "pep talk" at the right moments. Carlos and I have also closely collaborated in the exploration of tool support. Working with him has been pretty fun and I hope that we continue the cooperation in the future. A special thanks also goes to Tom Van Cutsem, whose expert knowledge on distributed programming and language design has been of great help to me. Tom has always found time to provide insightful comments on my ideas, papers and on parts of this dissertation. He also inspired me to investigate a reflective approach to language design.

I thank the members in my jury for the time invested reading my dissertation, and for their useful comments on this work: Prof. Franco Zambonelli (Università degli studi di Modena e Reggio Emilia), Prof. Siobhán Clarke (Trinity College Dublin), Prof. Bart Jansens (Vrije Universiteit Brussel), Prof. Viviane Jonckers (Vrije Universiteit Brussel), Prof. Beat Signer (Vrije Universiteit Brussel) and Prof. Ann Nowé (Vrije Universiteit Brussel).

All the past and present members of the Software Languages Lab have contributed to this thesis by creating a challenging and fun work environment, and by providing me with comments at research meetings. I am indebted to a number of people whom I have closely worked with during these years. Jorge Vallejos helped me to shape some of my initial ideas, and took over my teaching duties during the final stages of my writing. We also collaborated a lot (even though we did not share the same experimental platform). Thomas Cleenewerck stimulated me to explore research paths that I had not initially considered. Dries Harnie and I have shared many technical frustrations during our collaboration with MIVB in the context of our PRFB projects. His technical skills have also been of great help in the so-called legendary Android hacking sessions. He has also helped me with some teaching duties in the distributed programming course. Christophe Scholliers and Andoni Lombide Carretón deserve a special mention as my partners in crime within the Ambient group. Since they joined the lab, we have shared many hours of brainstorming, discussion, programming, writing papers and also drinking beers. Not only has our collaboration been fruitful from a research point of view, but it has helped me to regain my enthusiasm at difficult moments.

I would also like to thank all my office mates for the discussion and laughter: Jorge Vallejos, Peter Ebraert, Thomas Cleenewerck, Stijn Timbermont, Jessie Dedecker, Matthias Stevens, Stefan Marr and Dries Harnie. Also many thanks to our secretaries, Lydie Seghers, Brigitte Beyens and Simonne De Schrijver for helping me cope with the university administration, and for a chat from time to time.

Next to being a work environment, SOFT has also created a social environment where I have made a lot of friends. Many activities spontaneously organized at the lab have made my life in Belgium more enjoyable. Special thanks goes to the coffee drinkers, KK buddies and sport buddies, many of whom have been mentioned before, but there are also: Stefan Marr, Andy Kellens, Dirk Van Deun, Coen De Roover, Nicolas Cardozo, Yailen Martínez, Matthias Stevens, Isabel Michiels, and Kris Gybels. I also thank to the people that I regularly bother to get Dutch texts proofread.

I am equally indebted to a number of people who are not involved with my work, but who contributed in other ways to make this dissertation possible. Thanks to my friends from Barcelona for all the nice dinners each time I am in the country: Arnau Font Riera, Joan Parera Estapé, Anna Guinart Guri, Esther Payerols, Silvia Campoy Garcia, Ana Guitart Carrera, Cristina Santos, Eugenia Garcia Bordes and Emma Vázquez Martín. You make me feel as if distance and time do not matter. Special thanks goes to Ana for her unconditional support and listening ear that got me through moments of questioning, and Arnau and Joan, my dearest friends and partners in crime during my undergraduate days. Thanks to my Erasmus friends for all the nice meetings around Europe during these years: Clara Checchi Ponsa, Ruth Frutos Morales, Silke Van Wrangel, Thomas Kastner and Lea Katherine Acera. Also thanks to all my friends in Belgium for providing me with the necessary distraction. I am especially thankful to Virginia Gómez Oñate, Raül Romero Valls, Stefanie Vanosmael, and Kim Heymans.

Finally, I would like to thank my family. A special "thank you" goes to my parents and brother. There are not enough words to describe the support that my parents have given me. I am really happy that their dream of providing their children with higher education has become true. I am also deeply grateful to the Scholliers-Ceulemans family for their tremendous support and their warmth. They made me feel like one of them from minute zero. Also many thanks to Nathalie Scholliers who brought Twister into my life. Last but not least, I would like to thank my schat, best friend and future husband, Christophe Scholliers. He has supported me in every possible way by discussing ideas, proofreading parts of this dissertation, taking care of household chores during the final phase of my writing, convincing me to do sports or go out for dinner when I was stressed, etc. Most of all I thank him for just being himself and always believing in me. Love u!

This work has mainly been funded by a Prospective Research for Brussels (PRFB) project of Innoviris.

Elisa González Boix
October 2012

# Contents

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

In the latest years we have witnessed a hardware revolution that makes mobile devices get smaller and ever more powerful. For example, the latest smartphones are already dual core and incorporate various sensors including a compass, accelerometer, ambient light, GPS receiver, and even a barometer. In parallel, mobile devices have gained the ability to communicate using different wireless networking technologies with ranges as wide as 3G and WiFi, and as narrow as Bluetooth and NFC. This allows these computing devices to exchange information and interact without hindering but instead stimulating user mobility. In the last quarter of 2010, for the first time ever, smartphones already surpassed PC sales evidencing that mobile devices are becoming fully integrated into everyday activities supplanting existing computers as internet devices. This implies that ever more people become software users and being a software user no longer hampers one's personal mobility (as was the case with desktops and even laptops). Such a profusion of mobile devices bring us one step closer to Mark Weiser's *ubiquitous computing* era [Wei91].

Ubiquitous computing is a research vision which involves the presence of limitless computing power in everyday objects as small as coins, wrist watches, and microwave ovens, and as big as cars, buses and bus stops. The computer as a separate device thus disappears and its information processing capabilities are offered by means of intuitive interfaces embedded in the surroundings in an unobtrusive and invisible way. Technically, the mobility of wirelessly connected devices results in dynamically demarcated network topologies that are known as *mobile ad hoc networks* (MANETs). MANETs enable computers to interact with other devices that they encounter (as the user moves about) in a invisible way (as they use wireless technology), and are usually un-administered since these networks are formed spontaneously as a consequence of the collocation of mobile devices. MANETs have emerged as an enabling technology for ubiquitous computing and have been and are still subject of a large body of research in ubiquitous computing.

At the software level, this hardware revolution has opened up a whole new world of application possibilities. This resulted in a tremendous growth in the number of applications developed for mobile devices. While the Apple Store counts no less than 500,000 applications for the iPhone and iPad, the Google Play (previously known as the Android market) has reached a similar size achieving a threefold growth over the previous year. Despite the growth of the number of applications available on these mobile platforms, today, developing and testing distributed applications for mobile devices is still laborious. Moreover, most applications are actually just single device applications that do not fully exploit all the new networking possibilities. One of the main reasons is that programming languages that are commonly used for these tasks (e.g., C, C++, Java, Objective-C) have not been designed for the ubiquitous computing era. As a result, when developing distributed applications running on mobile devices, programmers have little more than a low-level socket API to work with, directly on top of the supported networking protocols.

This dissertation presents the results of our study of programming technology for supporting the development of MANET applications. In particular, we extensively explore the effects engendered by *partial failures* on software development technologies. We ground our research in the ambient-oriented programming paradigm [DVM$^+$06] developed at the Software Languages Lab. This paradigm has set the basis for a software platform that supports the development of applications running on MANETs, and identifies a set of programming language characteristics that explicitly take into account the hardware phenomena of mobile devices.

Within the context of this paradigm, the main goal of this dissertation is to propose expressive abstractions that aid developers to detect, reason about, and handle partial failures. A second goal of this dissertation is to explore tool support for debugging MANET applications in the face of partial failures.

## 1.1 Research Context

The research context of this dissertation is situated at the crossroads of several domains:

**Mobile Ad hoc Networks** The center of our research lies with a particular kind of distributed computing environments, known as mobile ad hoc networks (or just MANETs). A MANET emerges as a set of devices which become interconnected by means of wireless communication technology. The main characteristic of MANETs is that devices move about forming transient associations with other devices; associations which usually do not depend on any fixed infrastructure. As discussed above, our research focus is on the software development process for MANETs.

**Failure handling** The highly dynamic nature of MANETs exposes applications to a much higher rate of partial failures than traditional distributed computing environments. There exists no perfect functioning of the distributed system as such, but a graceful degradation of services as devices constantly appear and disappear from the network. This changes the methods and approaches that deal with partial failures, requiring new abstractions designed from the ground up to be used in a mobile setting.

**Programming language design** Among others, this dissertation places emphasis on programming language design. While raising failure handling to the programming language level may seem to make programming more difficult, we argue that it actually aids developers by providing a linguistic framework in which programmers can be made aware conceptually of the issues engendered by partial failures, and react to them accordingly. Since partial failures are so inherent to MANETs, we argue that they can no longer be hidden and be handled by the underlying software platform.

**Tool support** In order to support the construction of MANET applications, the software development process itself has to become more systematic. Software tools contribute to this task. This has motivated research in integrated development environments (IDEs) and other tools such as debuggers and profilers. Nowadays developers typically edit, compile and debug their programs in a single integrated environment. Distributed applications, in particular, MANET applications are not different in this regard. However, the omnipresence of failures in MANETs requires us to rethink the design and implementation of software tools. This work therefore also investigates tool support for MANET applications in the form of a debugger that handles partial failures.

This dissertation forms part of a larger research effort conducted at the Software Languages Lab in the field of distributed programming for MANETs. The main goal of this research group is to build sophisticated programming abstractions and tools that support the development of MANET applications. As such, this work can be seen as a continuation of the following earlier work:

- Dedecker in his dissertation on "Ambient-oriented programming" [Ded06] performs a first analysis of the implications of the hardware characteristics of MANETs at a software engineering level and proposes the *ambient-oriented programming* (AmOP) paradigm. It basically consists of a number of guidelines to be incorporated in future distributed object-oriented programming languages specially designed for MANETs. He builds a programming language called *AmbientTalk* to support experiments with AmOP language features.

- Van Cutsem's dissertation [Van08] further explored the AmOP paradigm to specifically reconcile traditional software abstractions for designating and communicating with objects in a MANET. This resulted in the design and implementation of *ambient references*. He also introduces *AmbientTalk/2* which supplants its predecessor while staying true to the fundamental characteristics of AmOP.

- In Lombide Carreton's dissertation [Lom11], the AmOP paradigm is used to support RFID-enabled applications. By applying AmOP to RFID-enabled applications, he is able to express RFID-tagged physical objects as software objects called *things*. He also investigates how to deal with the reactive nature of such objects on a new version of the AmbientTalk/2 language, called *AmbientTalk/R*, that automatically tracks dataflow dependencies to support reactive programming. Based on this AmbientTalk dialect, he builds abstractions for both node-centric and network-centric data flow programming.

- Finally, Vallejos dissertation's [Val11] investigates software modularisation techniques for MANET applications. He focuses on the modularity of two different concerns in the application's behaviour, namely, *context-dependent behaviour* (adaptation to contextual changes), and *group behaviour* (coordination of replicated services). For his experiments with modularization techniques, he builds *Lambic*, an incarnation of the AmOP paradigm in the generic function-based object model of Common Lisp.

This dissertation continues this tradition in AmOP research by presenting a systematic study of partial failures.

## 1.2 Problem Statement

*Partial failures* are a central ingredient of distributed systems. A partial failure occurs when one component of the distributed system fails, affecting a number of other components, and leading to a degradation of performance. When designing a distributed system, an important goal is thus robustness with regard to failures so that whenever a failure occurs the system can recover and continue working to some extent. As such, there exists a large body of software engineering techniques for making distributed systems fault tolerant. However, the bulk of this research is built on assumptions which no longer hold for MANETs. In general, it is assumed that communication is mostly reliable, the network is mostly connected, device failures are rare, and that, when they happen, devices do eventually recover and come back online. However, those assumptions no longer hold in MANETs because wireless communications are fragile and prone to fluctuations, and devices may appear and disappear from the environment at any moment as they move about. As a result, MANETs expose applications to a much higher rate of partial failures than traditional distributed systems.

As previously mentioned, the AmOP paradigm developed at our lab aims to provide a solution for this. Roughly speaking, it is a programing model in which network disconnections are treated as the normal mode of operation instead of representing them as exceptions. Remote object references are treated as "everlasting perfect" communication links that temporarily buffer messages in the case of network disconnections. Drawing an analogy with database transaction terminology, AmOP thus provides an *optimistic model* which makes network disconnections completely invisible to the programmer. Traditional distributed models, on the other hand, provide a *pessimistic model* in which network disconnections result in exception handling. Prior work as well as experience with students, has proven that this switch of model dramatically raises the abstraction level for writing MANET applications.

However, within this optimistic distribution model, the following problems remain to be solved:

**Lack of software development tools for MANET applications** So far, tool support has not been explored in the context of AmOP. Apart from traditional IDE support, we think that software tools, especially debugging tools, are vital to assist in the software development of MANET applications. However, debuggers for MANET languages and middlewares have received very little attention from academia so far. Even though research in the field of ubiquitous and mobile computing has been active for about two decades, this may be a symptom of the fact that the development of MANET applications has not matured yet.

A diverse spectrum of debugging tools does exist for concurrent and distributed systems. However, they lack mechanisms to enable debugging in the face of partial failures. In particular, they do not take into account that the target application is exposed to a high rate of partial failures. Bearing in mind the dynamic nature of MANETs, a debugging session will consist of an undetermined, fluctuating number of devices. As a result, a debugger must be able to allow devices to join and leave a debugging session without affecting the rest of the participants. Since many bugs may not manifest themselves until the application is actually deployed, debuggers should also be able to participate in running mobile systems.

**Lack of programming abstractions to deal with the effects caused by partial failures in MANET applications** As previously explained, adopting an AmOP paradigm leads to the optimistic programming model without failures. As a consequence, application logic is no longer polluted with exception handling code because the language's communication abstractions already take into account that the network connectivity is intermittent. Unfortunately, programming in such a "distributed programming paradise" engenders a completely new set of difficulties regarding network failures. First, not all failures can be resolved transparently by the programming language, and may require coordination among distributed parties. Second, and more importantly, not all failures are the result of transient network disconnections; permanent failures should also be dealt with. Because it is impossible to distinguish a transient from a permanent failure in a MANET, the unavoidable uncertainty has to be dealt with at the source code level. As a consequence, developers need to pollute application code with assumptions about the timing behaviour of the different distributed parties. Therefore, it is crucial to design a *failure handling model* that allows developers to understand and handle the various types of failures that may affect an application. In conclusion, we argue that the different nature of MANETs changes the methods and abstractions necessary to deal with partial failures.

## 1.3 Research Goals

In this dissertation, we study abstractions and tools for the construction of MANET applications that incorporate concepts to deal with the effects engendered by *partial failures*. As Dedecker pointed out in his dissertation [Ded06], "it is only when good software development technology becomes available that advanced applications will be developed". In the light of this observation, the goals of the research described in this dissertation are fourfold:

- We investigate programming language abstractions for failure handling in a MANET. To this end, we will propose a set of abstract criteria for a failure handling model to be used in a MANET. A failure handling model is central to enable programmers to detect, reason about and manage the effects of partial failures. Moreover, it provides a semantic representation of failures at the programming language level. We will use these criteria to study the state of the art distributed programming languages and middleware in Chapter 3. The conclusion of this study will reveal that combining a decoupled communication model with *leasing* provides a solution for devising a new failure handling model in MANETs. This will lead us to propose a concept that we call *ambient-oriented leasing*. This is mainly studied in Chapter 6.

- We investigate whether these failure handling abstractions can be integrated in a distributed model which is not necessarily based on the object-oriented programming style. Prior work on AmOP has formulated that AmOP as an extension to object-oriented programming. However, the outcome of our study of the related work will show that data-driven models such as tuple spaces or publish/-subscribe also exhibit some properties suitable for a mobile environment. As such, we want to show that it is also possible to reconcile the AmOP vision with a data-driven model. More concretely, we investigate how the aforementioned ambient-oriented leasing can also be integrated in a tuple space model. This is mainly studied in Chapter 7.

- In addition to the programming language abstractions for failure handling, we study tool support in the form of an *ambient-oriented debugger* that helps programmers to achieve a better understanding of the dynamic behaviour of a MANET application. It is our explicit goal to investigate debugging support in the face of partial failures *in tandem with* the aforementioned programming support for dealing with partial failures. This is mainly studied in Chapter 10.

- Finally, we investigate a novel distributed reflective architecture that makes possible the development of both programming and debugging support for partial failures in MANET applications. In the object-oriented research community, the entire Smalltalk school has come up with a *unified meta model* on which both the language as well as its tool support is built. Following this vision, we rethink the original meta-level model of ambient-oriented languages, and we propose the *transmitter-receptor* meta-level model. We embody the resulting reflective model in AmbientTalk/M, the language used to develop ambient-oriented leasing and the ambient-oriented debugger. This transmitter-receptor model is mainly studied in Chapter 5.

## 1.4 Research Methodology

To achieve the stated goals, this dissertation performs a vertical exploration of the effects of partial failures on the software development process. More concretely, we study the effects of partial failures on the design of a programming language, and then we move up to the supporting software development tools. We will achieve our goals in a *"proof by construction"* methodology in which we perform a number of design experiments. In the following sections, we motivate the main choices with regard to the development of these experiments.

### 1.4.1 A Language-oriented Approach

When designing a software platform to support partial failures in a MANET, a first design choice is how to offer the proposed abstractions, either as a programming language or as a middleware. Although the general trend in the field of mobile computing has followed a middleware approach, this work takes a language-oriented approach. This choice is motivated both by scientifically grounded considerations and by the expertise of our research lab. From a scientific point of view, prior research has described a number of reasons why a distributed programming language has advantages over a middleware or a library approach [BST89, VA01]. Bal et al. in [BST89] pointed out that the most important advantage of a distributed language is that it yields a higher level of abstraction. This is because the language provides *transparency* with respect to some concerns related to concurrency and distribution. Another important advantage pointed out by Varela and Agha [VA01] is the fact that distributed languages are better at enforcing certain properties of the underlying programming model by design. These advantages are exploited maximally in this dissertation. Moreover, the Software Languages Lab, in which this research has been carried out, has a long tradition of conducting language design experiments that are based on so-called "little languages" [Ben86]. Examples include a complete family of languages based on Pico [D'H96, DDD04], Agora [CDDS94], and AmbientTalk. The experiments conducted in this dissertation extend the family of ambient-oriented languages previously described.

### 1.4.2 The Limits of Transparency

From a historical point of view, research in distributed computing has been mainly driven by advances in the field of distributed object-oriented technology. Many research efforts have focused on hiding distribution behind traditional object communication abstractions. However, several influential papers have argued that a programming model should *not* provide the illusion of distribution transparency behind classical abstractions [GF99, WWWK96]. A better approach is therefore to recognize that distributed interactions have a number of discriminating properties which clearly set them apart from local interactions, the effects of partial failures being one of the most important ones. In this dissertation we endorse this vision.

Actually, our proposal for a distribution model suitable for MANETs will make failures explicit in the language. Given that failures are so omnipresent in a MANET setting, we aim to provide abstractions that help developers be aware of the effects caused by partial failures, and what to expect from the different parties in a distributed interaction when they can no longer communicate. However, even though we endorse the arguments against distribution transparency, we do not want to add unnecessary complexity to programming. Recurring patterns should therefore still be abstracted away as much as possible in programming language abstractions. As such, this work explores the trade-off between novel abstractions that aid with the difficulties of failure handling, on the one hand, and providing mechanisms that allow developers to be aware of their effects and take them into account in the design and construction of applications, on the other hand.

### 1.4.3 A Reflective Approach

There exist different methodologies for designing programming languages. We adhere to the vision that distinguishes programming language design from programming language implementation. While the language design process focuses on providing abstractions to ease programming applications, language implementation concentrates on techniques to efficiently provide those abstractions. Our research proposes some language abstractions that have been made available in a prototype implementation. Providing an efficient language implementation for them is outside of the scope of our research. This exploratory style of language design research is traditionally rooted in the realm of interpreter-based dynamically-typed programming languages. Such an approach has been previously adopted in high-level languages such as Smalltalk, Lisp, and JavaScript. These languages also have a rich tradition of being reflectively extensible. The main idea is that the language is based on a well-defined set of concepts that form the *kernel language*. An additional layer built on the kernel features then defines high-level abstractions to deal with certain aspects of an application (which in this work are related to distribution and failure handling). To this end, a *reflective* layer is introduced on top of the kernel making the language extensible from within itself. New language constructs can be reflectively added to the language as programming abstractions with syntactic support. Figure 1.1 depicts the design philosophy of a *reflectively extensible kernel language*.

Rather than building from the ground up a new ambient-oriented programming language, we have built our experiments by extending AmbientTalk. As previously mentioned, AmbientTalk was originally developed in the context of Dedecker's dissertation [Ded06]. It was mainly designed as a "language laboratory" for easing the

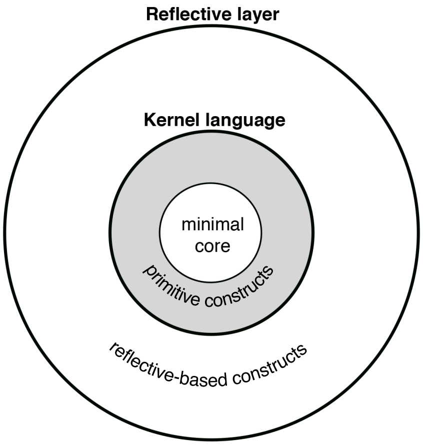


Figure 1.1: A reflectively extensible kernel language.

development and experimentation of novel language constructs for MANET applications. While the language proved to be a successful research vehicle, its object model and reflective architecture had a number of drawbacks (described in Van Cutsem's dissertation [Van08]). This motivated the development of a second incarnation of the ambient-oriented programming paradigm, AmbientTalk/2, which is at present the main distribution of AmbientTalk.

From a language designer point of view, an important difference between the two versions of AmbientTalk lies in the magnitude of the kernel language. As depicted in Figure 1.1, the kernel language can be further distinguished between a core and a set of primitive constructs "hardcoded" in the kernel. AmbientTalk/1 featured a minimal kernel building distributed constructs such as service discovery reflectively in the language. However, its reflective layer exhibits a number of conceptual problems, the most relevant being the lack of encapsulation and stratification between base and meta levels. AmbientTalk/2, on the other hand, features a more scalable kernel model, and a more modular and robust meta-level infrastructure. However, it adds more primitive constructs and it does not reify certain aspects necessary for the development of distribution and failure handling abstractions (as we will point out in Chapter 5).

In this dissertation, we take the design of the reflectively extensible kernel language one step further and we revisit the AmbientTalk/2 kernel language in order to provide a minimal kernel by reducing the number of primitive constructs, favouring the construction of language constructs for distribution reflectively. The resulting language, called *AmbientTalk/M*, will be then exploited to build both our failure handling abstractions, and our ambient-oriented debugger.

## 1.5 Technical Contributions

In this section, we summarize the main contributions of this dissertation:

- We define a *set of criteria* which identify the characteristics that a failure handling model must possess in order to be used in a MANET (cf. Table 2.1). Part of these criteria are based on the definition of ambient-oriented programming, which we extend to better cope with the effects engendered by partial failures.

- We propose meta-level engineering to support better the development of distribution and failure handling abstractions and tool support. In particular, we revisit the meta-level engineering of an ambient-oriented language and introduce two new concepts: a novel reflective model to represent remote object references, and a technique that enables meta-programs to *dynamically* react to the manipulation of objects by the interpreter. We embody those concepts in a new dialect of AmbientTalk/2, called *AmbientTalk/M* (cf. Chapter 5).

- We present an approach to a failure handling model for MANETs based on the notion of leasing. We identify a set of criteria for integrating leasing at the heart of a failure handling model designed for a MANET. We then define a notion of a lease which we embed into the abstraction of remote object references, giving rise to a novel kind of distributed referencing abstractions called *leased object references*. We show a number of language abstractions built around leased object references in order to decrease the programming effort of leasing. We finally provide leased object references as an extensible framework that allows programmers to express their own leasing patterns.

- We provide a second embodiment of ambient-oriented leasing based on tuple spaces. This results in a novel kind of tuple space model called *TOTAM* ("Tuples In The AMbient") which integrates leasing into a replication-based tuple space model. Chapter 7 describes its design, concrete instantiation in AmbientTalk, and a first formalization of the model by means of an operational semantics.

- We promote the development of ambient-oriented applications by proposing the features of an *ambient-oriented debugger*. As a proof-of-concept implementation, we provide an online ambient-oriented debugger integrated into AmbientTalk Eclipse IDE called *REME-D* (Reflective Epidemic MEssage-oriented Debugger). This prototype tool shows that it is feasible to implement an ambient-oriented debugger using a mirror-based reflective architecture with the enhancements introduced in Chapter 5.

### 1.5.1 Supporting Publications

Of the (co-)authored publications that are related to mobile computing [MDG+06, VVG+07, VMG+07, GVV+07, MVT+09, GSL+10, SGBDMD10, GBLCS+11, GBCVC+11, BVB+12, HGDD12], the following introduce the key ideas of this dissertation.

- **Mirror-based Reflection in AmbientTalk** [MVT+09]
  Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Eric Tanter, Wolfgang De Meuter
  *Software: Practice and Experience*, 2009
  This paper proposes a mirror-based reflection architecture which reconciles mirror with behavioural intercession in the context of AmbientTalk. It provides the first ad hoc integration of leasing as a remote object reference, resulting in an ad hoc implementation of leased object references. Based on our experiences with such a reflective architecture, we will build AmbientTalk/M (cf. Chapter 5).

- **A Leasing Model to Deal with Partial Failures in Mobile Ad hoc Networks** [GBVCJ+09]
  Elisa Gonzalez Boix, Tom Van Cutsem, Jorge Vallejos, Wolfgang De Meuter, and Theo DHondt
  *In the 47th International Conference on Objects, Models, Components, Patterns (TOOLS 2009)*
  This paper proposes the core ideas of the leasing model described in Chapter 6 in which leasing is combined with a decoupled communication model in an object-oriented manner. It also describes an earlier version of the programming and implementation API for leased object references described in this dissertation.

- **Context-Aware Tuples for the Ambient** [SGBDMD10]
  Christophe Scholliers, Elisa Gonzalez Boix, Wolfgang De Meuter, and Theo D'Hondt
  *In the 12th International Symposium on Distributed Objects, Middleware, and Applications (DOA) at On the Move to Meaningful Internet Systems (OTM 2010)*
  This paper presents the core ideas of the TOTAM tuple space model described in Chapter 7. The paper focuses on the software engineering support that TOTAM provides for context-awareness.

- **REME-D: a Reflective Epidemic Message-Oriented Debugger for Ambient-Oriented Applications** [GBCVC+11]
  Elisa Gonzalez Boix, Carlos Noguera, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt
  *In the 26th Annual ACM Symposium on Applied Computing (SAC 2011)*
  This paper presents REME-D, our online ambient-oriented debugger for AmbientTalk applications. In particular, it presents an earlier version of the REME-D prototype tool presented in Chapter 10 which was mostly reflectively implemented, but still required some modification on the underlying interpreter.

## 1.6 Dissertation Roadmap

This dissertation has been structured in two different parts corresponding to the two main research problems this research addresses (described in Section 1.3). We now summarize each chapter in this dissertation.

**Chapter 2: Taxonomising Partial Failures in Mobile Ad hoc Networks** proposes a set of criteria for a failure handling model for dealing with partial failures in a MANET. The chapter starts by describing the hardware characteristics of MANETs which serve to motivate our criteria. We then provide some basic terminology for failures and identify three key dimensions in the design of a failure handling model. Subsequently, we define a set of criteria for failure handling models to be suitable for use in MANETs according to these dimensions. Finally, we revise our criteria in the light of facilitating the development of software development tools for MANET applications.

**Part I** focuses on exploring the effects of partial failures in MANET applications, and introducing suitable programming abstractions for easing their construction.

**Chapter 3: Related Work** primarily surveys a number of distributed programming languages and middlewares in the light of the criteria identified in the previous chapter. As mentioned before, the outcome of this survey is that combining a decoupled communication model with leasing offers a good basis for providing a failure handling model in MANETs. However, to date leasing has been made transparent to applications or very low-level support is provided to manipulate leases. The main goal of this part is to integrate a leasing model into ambient-oriented programming by means of appropriately designed language support, leading to the concept of *ambient-oriented leasing*.

**Chapter 4: Ambient-Oriented Programming in AmbientTalk** introduces the AmbientTalk programming language, an example of the ambient-oriented programming principles on which the work of this dissertation builds. We describe those language features required to understand the technical details of following chapters. In particular, we focus on concurrent, distributed and reflective language features.

**Chapter 5: Enhancing Meta-level Engineering in AmbientTalk** discusses the shortcomings of AmbientTalk's reflective architecture for the development of referencing abstractions and software development tools for ambient-oriented applications, motivating the need for a revisited meta-level engineering. Subsequently, we introduce AmbientTalk/M, a dialect of AmbientTalk which revisits the object and actor reflective layer and introduces the following new concepts: a new representation of remote object references as a transmitter-receptor pair representing both ends of a reference, and an observer mechanism that enables meta-programs to dynamically react to the manipulation of objects by interpreter. Such a reflective architecture is subsequently employed in Chapter 6 to implement our language abstractions for ambient-oriented leasing, and in Chapter 10 to implement an ambient-oriented debugger.

**Chapter 6: Ambient-Oriented Leasing** investigates how to integrate leasing at the heart of a failure handling model suitable for MANETs. The chapter starts by identifying three characteristics required for a leasing model to be usable in a MANET. We then present a detailed definition for our notion of a lease, and integrate it into distributed communication leading to the abstraction of *leased object references*. We then explore the integration of our lease concept into distributed computation by means of *due-type messages* which enable to control delivery guarantees of asynchronous messages exchanged between distributed parties. Our approach is illustrated by a running example of a mobile music player application in which we put our abstractions to work. We also investigate the effects of these two new language abstractions from a software engineering perspective, and introduce support to alleviate the programming effort they introduce. Subsequently, we describe the implementation of leased object references as an extensible framework in which custom leased reference variants and leased-based abstractions can be expressed. We evaluate our language constructs by comparing our implementation of the music player application with an implementation using Java RMI.

**Chapter 7: Ambient-Oriented Leasing for Tuple Spaces** integrates ambient-oriented leasing in a tuple space model, resulting in a novel adaptation of the tuple space model for MANETs, called TOTAM (Tuples On The AMbient). We describe the TOTAM tuple space model and a concrete prototype implementation in AmbientTalk. We present a practical API for TOTAM and show its use by means of the implementation of a mobile game. We also provide an operational semantics for our model.

**Chapter 8: Ambient-Oriented Leasing Under The Microscope** concludes the first part of this dissertation by discussing ambient-oriented leasing in the light of the criteria for a failure handling model postulated in Chapter 2.

**Part II** focuses on exploring software developments tools to support the development of MANETs in the form of a debugger.

**Chapter 9: Related Work** starts the second part of this dissertation by surveying the state of the art in distributed debugging tools and techniques.

**Chapter 10: Debugging in the Face of Partial Failures** focuses on providing debugging support for ambient-oriented applications. The chapter starts by discussing the challenges of debugging ambient-oriented applications, and presents the main features of an *ambient-oriented debugger* to address them. We then describe the design and implementation of a concrete instance of those features in *REME-D*, a Reflective Epidemic MEssage-oriented Debugger. We present REME-D as an online ambient-oriented debugger for AmbientTalk that integrates techniques from traditional sequential and distributed debuggers into a decoupled communication model, and proposes novel facilities to deal with hardware features of MANETs.

**Chapter 11: Pre-experimental User Study for REME-D** validates our debugger by conducting a pre-experimental user study which follows a one-group pretest-posttest quasi-experiment design. We subsequently discuss the insights gained with regard to how real users perceived and valued REME-D's features.

**Chapter 12: Conclusion and Future Work** concludes our dissertation. We revisit the problem statement and highlight the contributions of our dissertation with hindsight. Finally, we discuss interesting avenues for future research.

# Chapter 2

# Taxonomising Partial Failures in Mobile Ad hoc Networks

Partial failure is one of the key issues that distinguishes distributed from parallel computing. In fact, several authors have considered partial failure to be the defining characteristic of a distributed system, quoting Leslie Lamport [Lam87]: "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." Hence, robustness against partial failures has been a main requirement in the design of distributed systems. Yet, a large body of research in fault tolerance focuses on hiding failures from the application layer and supporting high availability (through replication). This is justified because, traditionally, failures have been considered to add complexity to programming; so, the more a distributed system resembles a sequential one, the easier it is to program it. Gerraoui and Fayad already pointed out the dangers of providing the illusion of distribution transparency, which they refer to as the *myth of transparent distribution* [GF99]. Nowadays, researchers are aware of the irreconcilable differences between sequential and distributed computing. However, in general, it is assumed that a distributed system works perfectly until some "catastrophic" failure happens which causes a node to stop working completely, leading to a degradation of *performance* [Nik00]. This assumption no longer holds for applications running on mobile ad hoc networks (MANETs). Due to the highly dynamic nature of MANETs, devices may appear and disappear from the environment at any moment. As such, there exists no perfect functionality of the system. Rather, fault tolerance implies graceful degradation of *services* and dynamic adaptation to the changing environment. Next to changing the concept of fault tolerance, the different nature of MANETs also changes the methods and approaches to deal with partial failures. Basically, MANET applications must be written assuming that failures are the rule rather than the exception.

The approach that we take in this work is to deal with partial failures at the programming level by uncovering suitable failure handling abstractions which are appropriate for use in MANET applications. In this chapter, we first characterize the problems of partial failures in MANET applications. Subsequently, we distill a set of criteria to adhere to in order for a failure handling model to be suitable for MANETs. Such criteria are grounded in the ambient-oriented programming paradigm [DVM$^+$06], which in itself already identifies a number of requirements for the development of MANET applications.

## 2.1 Mobile Ad hoc Networks

Mobile ad hoc networks (MANETs) are computer networks that are spontaneously formed when a number of *mobile* devices that are connected by means of *wireless* technology. Quoting Murphy et al. in [MRV98] a mobile ad hoc network is “a transitory association of mobile nodes which do not depend upon any fixed support infrastructure”. Such a network thus emerges due to the collocation of mobile devices, resulting in opportunistic wireless interactions amongst the devices.

Depending on the type of mobile devices and wireless networking technologies used, there may exist different types of MANETs. On the one hand, mobile devices can vary between, laptops, tablets, cellular phones, and other electronics embedded into items such as car’s on-board computers, sensor nodes, or RFID-tagged objects. On the other hand, wireless technology can range from Wifi technology, Bluethooth, NFC, to 3G mobile internet. As a result, interactions amongst devices typically happen over fragile communication links as disconnections of devices can happen at any moment in time due to the limited network coverage of wireless technology. The types of MANETs that this work mainly focuses on are *wifi-based MANETs* in which mobile nodes are interconnected by Wifi technology (such as Wifi-direct) which does not rely on Internet-gateway access points. We also consider *vehicular ad-hoc networks* (VANETs) in which mobile nodes can interact and cooperate with “mobile infrastructure” (such as vehicles) and other stationary infrastructure (such as roadside equipment) which is interconnected via Wifi. The devices that we target are powerful mobile devices such as tablets or smartphones. As such, we will not target mobile ad hoc networks composed of thousands of small mobile devices with limited computational power such as sensors networks.

MANETs can be used to deploy a broad spectrum of applications, which we refer to as *MANET applications*. We are particularly interested in the software development of MANET applications that range from *collaborative applications* [KB02] in which a number of devices interact in impromptu meetings to *urban-area applications* [HGDD12] in which mobile users connect with various access points of a mobile infrastructure which is itself constantly in motion. Examples of collaborative applications include distributed drawing editors, instant messengers, file and music sharing applications, and mobile social networking applications [GBLCS+11] (that exploit the collocation of users to shape social interactions). On the other hand, examples of urban-area applications include city games like Hitchers [DBT+06], peer-to-peer transport information applications in which commuters can obtain real-time traffic information of their itinerary from vehicles (e.g., buses, trains, etc.) and even places (e.g., bus stops, traffic lights, etc.), vehicular communication applications in which cars communicate with one another to avoid traffic jams, etc.

### 2.1.1 Hardware Characteristics

As previously mentioned, mobile ad hoc networks are composed of *mobile* devices which communicate with each other by means of *wireless* communication links with a limited communication range. Van Cutsem et al. [VMG+07] identified two discriminating properties which clearly set MANETs apart from traditional fixed networks:

**Volatile Connections** Mobile devices equipped with wireless technology possess only a limited communication range. This combined with the fact that users move about with their devices implies that communicating devices may move out of communication range at any time without notice. The resulting disconnections are not always permanent: the two devices may meet again at some point later when able to reconnect. Often, such *intermittent* network connections should not affect an application, allowing communicating parties to continue their collaboration where they left off. Because transient disconnections are omnipresent, a disconnection should no longer be treated as a “failure” by default in MANETs.

**Zero Infrastructure** Mobile ad hoc networks are formed by the temporary collocation of mobile devices. As a device moves about, it will spontaneously join with and disjoin from the ad hoc network. As a result, there is very little or no fixed infrastructure on which devices can rely to discover and collaborate with one another, e.g., a name server to manage service discovery such as Java RMI registry. The services available to applications have to be dynamically discovered when they become available on proximate devices. As such, MANETs are usually un-administrated.

Although MANET applications have been previously identified as a subset of ad hoc applications running on MANETs [Van08], they introduce a number of challenges at the software level because *any* application has to deal with the aforementioned hardware characteristics. These hardware characteristics are *universal* as they affect and often pervade the entire application independently of the functionality it provides. As a result, they undermine the assumptions made by long-established methods, algorithms, and technologies for distributed computing.

## 2.2 Ambient-Oriented Programming

The ambient-oriented paradigm (AmOP) was specially designed to help developers with the development of a MANET applications. The paradigm proposed a set of well-defined guidelines that a system must comply with in order to be suited for the mobile environment:

**Classless Object Model** Due to the dynamic nature of MANETs, an ambient-oriented language requires objects to be self-sufficient: code and data should be integrated together in an object, rather than separating the code in a class. When distributing objects across a network in a class-based language, objects and their classes are copied between devices. As a result, objects residing on different machines can autonomously update their class even though, conceptually, there is only one class. To avoid a distributed state consistency problem among duplicated classes, ambient-oriented languages disallow the use of classes, favouring prototype-based programming instead.

**Non-Blocking Communication Primitives** The volatile connection phenomenon inherent to MANETs requires communication primitives to be *non-blocking*: the process sending a message should not be suspended while completing the operation. As a result, when a communicating party moves out of communication range, no other concurrently running party will ever be blocked. This approach minimizes the effect of temporary unavailability of devices [MLE02], and avoids potential distributed deadlocks.

**Reified Communication Traces** Since communication is non-blocking, both senders and receivers continue their execution regardless of message sending or reception. This means that synchronization is necessary to prevent communicating parties from ending up in an inconsistent state. To resolve these inconsistencies, an ambient-oriented language should store an *explicit representation* (i.e. a reification) of the communication details of processes. This allows a party to properly recover from an inconsistency by reversing (part of) its computation.

**Ambient Acquaintance Management** In order to deal with the zero infrastructure phenomenon previously described, an ambient-oriented programming language should allow an object to spontaneously get acquainted with a previously unknown object without knowing its address beforehand. Ambient acquaintance management also implies that communicating parties must be able to keep an up-to-date view of which acquaintances are (dis)connected so that they can take explicit action when an acquaintance disconnects.

These four language design principles advocate the explicit incorporation of the hardware characteristics of MANETs described in the previous section in future distributed object-oriented languages. However, while the paradigm incorporates simple semantics for handling intermittent disconnections as a result of the volatile connections phenomenon, it says very little about how to deal with the effects engendered by partial failures. Quoting Dedecker in his dissertation [Ded06]:

> "Although a paradigm is a first necessary step towards supporting the design and development of AmOP applications it does not necessarily give insight into how such applications are built and how different (distribution) concerns should be expressed.".

*Within the context of ambient-oriented programming, this dissertation specifically explores how failure handling should be dealt with, in particular, how to detect, reason, and handle partial failures at the programming level.* While raising failure handling to the programming level may seem to make the programming more difficult, we argue that it actually aids developers by providing a framework in which they can be aware of the issues engendered by partial failures, and what to expect from the different kinds of distributed interactions. Before defining criteria for failure handling abstractions in MANETs, we review the types of failures that can hinder interactions in MANETs.

## 2.3 Types of Failures

In order to understand the importance of a failure handling model for MANET applications, we first define basic terminology related to failures and we explain what it means to tolerate failures.

A *failure* is traditionally defined as an event that occurs when the services provided by a system deviate from the ones it was designed for [ALRL04]. In the case of distributed systems, a failure implies that one or more services offered by the system cannot be provided. The discrepancy between the observed behaviour and the theoretically correct behaviour of a system is called an *error*. Hence, an error is an event that may lead to a failure. For example, errors when transmitting a data package across the network may lead to a failure if the client process is unable to deserialize the package. Finally, a *fault* is an incorrect step in a program which causes an error (e.g., the cause of a transmission error may be a deteriorated network cable). A fault is said to be *active* when it causes an error, and *dormant* when is present in a system but latent. Avizienis et al. in [ALRL04] describe the relationship between faults, errors and failures as the "chain of threats". The activation of a fault results in an error either because an internal dormant fault becomes active or due to an external fault. The propagation of such an error to the service interface then makes the service deviate from the correct behaviour, resulting in a failure. The failure of a service in turn causes an external fault for the system receiving the service completing the cycle of fault, error and failure.

A *failure model* (also called the system's failure modes) defines the different types of failures that a system tolerates. There exists several classification schemes in the literature which provide insight into the different types of failures that affect distributed systems [TS01, ALRL04, GR06]. Based on those schemes, this section provides a precise description of the kind of failures that are relevant in a mobile distributed system. We classify failures according to three different aspects: scope of the failure, cause of the failure and the duration of a failure. These concerns are depicted in Figure 2.1.

The term "process" denotes a software entity that has autonomous behaviour, internal state and which interacts with other processes by exchanging messages through the network. A distributed system is thus a collection of processes that communicate with one another to provide some services.

**Scope of the failure** As a first basic distinction, failures can be either *total* or *partial*. A total failure happens when all the software entities working in the system cease to compute. Failures are total in sequential programming where they typically result in an exception or the inability to complete the computation. In contrast, in distributed computing, the failure of a process may not affect the correct functioning of other processes in the network. Hence, failures in a distributed system are said to be partial. Although a total failure is rare in distributed systems, it is still possible e.g., in systems deployed in highly static network topologies interconnected by reliable technology such as Ethernet. This is not the case in a mobile distributed system: because of the mobility of devices and the unreliability of wireless technology, partial failures are inherent to MANETs.

**Cause of the failure** In a distributed system, failing to provide services means that either processes, communication channels, or both are not doing what they are supposed to do. According to the entity which is deemed to cause the failure, we distinguish between process failures, communication failures, and arbitrary failures. Determining what caused a failure is important so that the system can apply compensating actions and continue working.

*Process failures*: They form the simplest way of failing; they occur when a process stops executing some computation, and does not send any message to other processes. Process failures can be caused by programming errors, security attacks, or hardware malfunction. They may affect the whole process or only certain components (or threads) within the process. A process failure can be further refined depending on whether other processes can reliably detect it or not [GR06]. When a process stops all computation and notifies this to other processes, the failure is said to be a *fail-stop* failure. When a process stops without notification, the failure is said to be a *crash* failure. In a mobile distributed system, processes typically do not fail gracefully notifying other processes as many failures are unanticipated because of the adhocness of the environment. Since other processes do not get notified of the failure, they just perceive a process failure as a lack of responsiveness. As a result, process failures cannot be

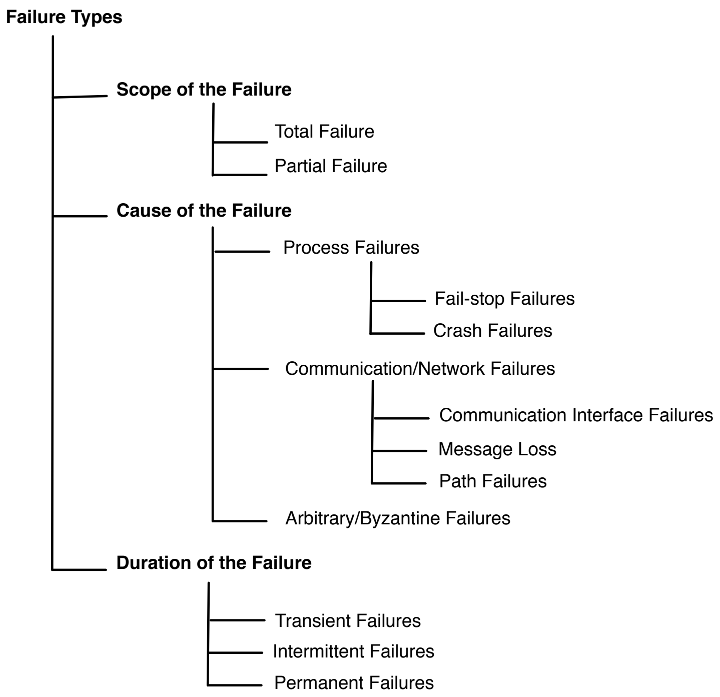


Figure 2.1: Classification of failures.

distinguished from the failure of the communication channels. This brings us to the definition of the second type of failures within this category.

*Network failures*: They arise when the physical communication medium that connects the process with the rest of the system does not transport messages anymore. Network failures are also referred to in literature as failures in communication, or just *communication failures*. In this dissertation, we use these terms interchangeably. A network failure can be subdivided in three categories [DMQ07]:

- *Communication interface failures* occur when the network interface transmitting outgoing messages fails, or when the receiver's interface fails to process incoming messages, or when both interfaces fail. Hadzilacos and Toueg further distinguish between *send-omission failures* and *receive-omission failures* [HT94]. The former happens when a process completes the transmission but the message is not put in its outgoing message buffer. The latter happens when a message is put in the incoming message buffer, but the process does not receive it. In this work, we use communication interface failures in the broadest sense of the term including send-omission and receive-omission failures.

- *Message lost* implies that individual messages transmitted by a process may be dropped or lost on the communication route between the different nodes in the network. The loss of messages in a communication channel has also been termed *communication omission failures* [CDK05].

- *Path failures* occur when a communication channel between two nodes appears to be blocked. Because the effects of path failures are similar to communication interface failures, we only consider communication interface failures.

In traditional distributed systems, network failures are usually either forced for administrative reasons (e.g., hardware or software upgrades) or caused by an unpredictable malfunction (e.g., overloading, interference, congestion of the network, denial of service attacks, physical detach of cables, etc). These assumptions do not hold for mobile distributed systems because of several reasons. First, MANETs are typically un-administered (so network failures are not forced for administrative reasons). Second, devices move about. As a result, many network failures are spontaneously caused by a lack of network coverage when a device moves out of communication range and leaves the network. Finally, many failures are simply voluntarily caused by the end user, e.g., to save battery power, optimize the available bandwidth, maximize the mobile data connection, etc. End-users thus play a crucial role in mobile distributed systems; their arbitrary mobility makes unpredictable network failures the norm.

*Arbitrary failures (also known as Byzantine failures)*: They are the most general case of failures. They occur when the system fails in arbitrary ways because of software bugs or malicious behaviour. Those failures do not follow any particular pattern: processes may stop, crash or even continue working producing incorrect or inconsistent computation (e.g., sending messages that have nothing much to do with those supposed to be sent regarding the application functionality). Due to their unpredictability nature, they may encompass crash failures, communication interface failures, message loss, as well as malicious failures (e.g., corrupting local state, processing or sending incorrectly a message, etc).

Arbitrary failures are very harmful to MANETs because they may cause damage in unanticipated ways. Since devices generally form transient associations which do not depend on any fixed infrastructure, it is really difficult to detect the compromised devices. Some work in MANETs has focused on making routing protocols resilient to such failures [PHM06, ACH$^+$08, GPP$^+$10]. This work aims to tackle malicious disruptions of the data transmitted over a wireless link (i.e., ensure *secure data communication*), and the correctness of the routes the data is sent across (i.e., ensure *secure route discovery*). However, there has been little research on generic failure detection mechanisms for them [Col07]. Quoting Doudou et al. in [DGG02]:

> “Byzantine failures have many faces, some of which can simply not be encapsulated inside the failure detector.”

Detecting arbitrary failures requires complementary security-related mechanisms such as access control techniques, cryptographic key management, etc. In this work, we assume that processes and communication channels are not subject to arbitrary failures. Dealing with arbitrary failures requires the design and implementation of programming abstractions for enabling secure interactions among potentially malicious devices while minimizing the possible damage, which is outside the scope of this dissertation.

**Duration of the failure** The last concern classifies failures according to the possibility of recovery: a failure is said to be *transient* if the system recovers after a finite amount of time; otherwise it is said to be *permanent*. Tanenbaum et at. in [TS01] further redefine transient failures according to how often they occur. Transient failures occur once and then disappear, while *intermittent* failures denote failures that appear, then disappear suddenly, then reappear, and so on. In mobile distributed systems, many transient failures are intermittent because of the hardware characteristics of wireless technology and the mobility of devices. In this work, we generally use these two terms interchangeably to refer to failures in which the system recovers at some point in time.

Due to the lack of global knowledge in distributed systems, it is hard to determine whether a failure is transient or permanent. This problem is exacerbated in mobile distributed systems since a failure can be caused by a process failure, by a network failure, or simply because the user left the network. As Waldo et al. already remarked in [Wal01], only the future can prove a failure to be transient or permanent. However, since the time that the system needs to wait for a connection to be reestablished can depend on device mobility and end-user preferences, it is impossible to ascertain automatically whether a failure is transient or permanent. As a consequence, the way we represent and expose failures in the software platform is crucial to help developers deal with such undeterminism when building MANET applications. The next section describes a set of requirements that we use to determine what kind of failure handling abstractions are suitable for mobile distributed systems.

## 2.4 Design Dimensions of a Failure Handling Model

In the previous section, we established what types of failures are relevant in the context of MANETs. As we explained, the hardware characteristics of MANETs change the assumptions about the properties of the underlying communication network. This, combined with the fact that devices move about makes that applications are exposed to a higher rate of partial failures in MANETs than in traditional fixed networks. The lack of adequate support to deal with failures puts an extra burden on developers which need to deal manually with their effects at the application level. Since failures in MANETS are unpredicatable and omnipresent, a mobile distributed system should be able to detect failures and react to them in a consistent way.

We term a *failure handling model* the set of abstractions and software techniques for detecting and reasoning about partial failures. A failure handling model is a crucial abstraction in a software platform designed to build applications running on MANETs because it makes developers aware of the hardware environment in which applications run. In addition, it also make them take into account the indeterminism inherent to MANETS during the application design. The design of a failure handling model has a major impact on the design of the software platform as a whole. In particular, it affects at least the following three components:

**Communication** As previously mentioned, the lack of communication of a process in a distributed interaction is the usual symptom of a partial failure. Since the failure of a process is indistinguishable from a network failure in MANETs, detecting failures usually involves communication. Failure detection can happen actively (processes send "alive" messages to each other) or passively (processes wait for the arrival of messages from other processes). At the language level, the failure handling model may be completely aligned with the communication facilities (e.g., Rover [JdT$^+$95] augments the remote method procedure (RPC) abstraction with queueing capabilities to cope with intermittent disconnections), or it can be made an independent abstraction (e.g., in the language Oz [HRBS98] developers can install fault streams on software entities in order to react to failures). In short, a failure handling model must allow process to detect failures when communicating with other process.

**State consistency** A fundamental aspect of failure handling is to be able to bring the system back into a consistent state once a failure has occurred. A failure handling model must provide abstractions that allow processes to protect against failures and recover from them in a consistent and reusable way. In traditional distributed systems, protection against failures is achieved using redundancy techniques that mask failures and support continued operation of the system by enhancing the availability of data (as in the case of replication, and distributed transactions). Recovery is typically achieved by regularly saving the system state, and get back to a previously correct state using checkpointing or message logging techniques.

**Memory management** During the lifetime of an application, processes will interact and share resources with an unknown number of processes. Moreover, the process exporting a resource to the network may not necessarily be active at the same time as the one using it. This may lead to an accumulation of outdated information, compromising the performance of the application, and even the mobile device. A proper failure handling model should be devised along with the memory management scheme to limit the usage of resources shared with other processes.

In short, *a failure handling model involves a set of programming abstractions that address the above aspects of a distributed model.* Note that a wide range of approaches have been proposed to address some of these aspects. Most of the existing works have focused on making one of these aspects resilient to partial failures (e.g., much research has focused making communication facilities like RPC tolerant to failures). Some programming models for traditional distributed systems incorporate a failure handling model which takes into account all three aspects (as in the case of the Oz language which we discuss in the next chapter). Before analyzing the various approaches in state of the art of software development platforms, we present a set of criteria to evaluate their applicability in a MANET.

## 2.5 Criteria for a Failure Handling Model for MANETs

We now describe criteria that define which failure handling abstractions are suitable for use in MANETs. The criteria are organized according to the previously identified dimensions in the design of a failure handling model: communication, state consistency, and memory management.

### 2.5.1 Partial Failures and Communication

In any distributed system, processes must be able to discover other processes, send and receive information to and from other processes, and synchronize with one another to provide a number of services. The term *communication* comprises these three activities. In the literature, *coordination* is also used to denote the communication concerns of a distributed system [PA98]. We adopt a loose interpretation of these terms, and we sometimes use the two interchangeably.

Traditional distributed models usually provide means to detect the failure of a communication primitive, and represent such a failure to the application level by propagating exceptions. Due to the volatile connections phenomenon previously described, communication between devices is expected to be interrupted, often for unpredictable amounts of time. This results in application code that is polluted with exception handling code because failures are the rule rather than the exception in MANETs. We therefore advocate a communication model which abstracts over intermittent failures *by default*. This is the essence of our first criteria which is derived from the notion of ambient-oriented programming presented in Van Cutsem's dissertation [Van08].

**Criterion 1 (Decoupled communication)** Processes should be decoupled as much as possible in space, time, synchronization, and arity to allow computations to proceed in the face of the intermittent failures and zero infrastructure phenomena present in MANETs.

Communication is decoupled in time when processes do not need to be active at the same time to communicate with one another. It is decoupled in space when processes do not need to know each other beforehand to communicate. Those two forms of decoupling (or uncoupling) were originally remarked by Gelernter in his work on generative communication [Gel85]. Eugster et al. in [EFGA03] further distinguished synchronization decoupling to denote that the control flow of communicating processes is not blocked upon sending or receiving of a message. Van Cutsem also points out the importance of the arity decoupling in a MANET. Arity decoupling states that processes do not need to know the total number of processes communicated with.

It is important to remark that these forms of decoupling are not restricted to a particular programming paradigm. In general, distributed programming models can be categorized into *data-driven* and *control-driven*[1]. In a data-driven model, communication takes place by describing the characteristics of *data* items exchanged by an intermediate coordinator. Representative examples that instantiate such a model are tuple spaces, and publish/subscribe systems. In those approaches, the coordinator takes the form of an associative data structure (or shared dataspace), and an event broker (or event service). In a control-driven model, communication happens instead by means of well-defined interfaces that interconnect processes, usually referred to as *communication links*. Processes behave as black boxes which use communication links to convey information to other processes. Distributed object-oriented systems are the classic example of a control-driven model in which processes communicate by means of remote object references. Note that communication links are not restricted to point-to-point interactions. Other kinds of interactions are definitely possible such as group communication (by forming a 1-n relationship between processes), and peer-to-peer architectures (by organizing processes into an overlay network). Although much research in distribution has focused on object-orientation, data-driven models like tuple spaces and publish/subscribe have been adapted to exhibit some forms of decoupling. We further discuss them in the next chapter.

A distributed model adhering to the decoupled communication criterion is attractive since it provides simple semantics to deal with transient failures in a MANET. However, permanent failures should be handled as well. In the previous section, we already remarked that it is impossible to accurately distinguish a transient from a permanent failure in a MANET. Since failures can only be approximated, the unavoidable uncertainty has to be dealt with at the application level. This implies that developers need to make assumptions about *the timing behaviour* on communication between processes. This is not straightforward and it can depend on system parameters such as the number of clients. This problem is exacerbated in MANETs due to the unpredictable mobility of devices. Therefore, it is crucial to complement the decoupled communication model with dedicated support for encapsulating information about a failure. We

[1] This terminology is based on the taxonomy of coordination models proposed by Papadopoulos in [PA98] which can be equally applied to distributed programming models.

explicitly state the importance of an explicit representation of an application's timing assumptions in the following criterion.

**Criterion 2 (High-level representation of failures)** Processes should agree on criteria to determine when their *logical* communication has terminated.

Agreeing on some criteria to delimit their communication allows processes to autonomously differentiate transient from permanent failures and know what they can expect of each other when they are unable to communicate. This means that communicating parties must rely on some mechanism other than network connectivity to detect failures. In the literature, we observe that failure detectors have been introduced as a dedicated abstraction to encapsulate time assumptions [CT96]. A failure detector can be regarded as an "oracle" that provides information about processes that have failed. However, failure detectors have been designed only for synchronous and partially synchronous systems[2]. In contrast, mobile distributed systems are asynchronous by nature. Chandra and Toueg showed in [CT96] why it is impossible for a failure detector to achieve both *completeness* and *accuracy* over an asynchronous system. Completeness requires that a failure detector eventually suspects every process that actually crashes, while accuracy requires that there is a time after which a failure detector does not make a false detection. This observation has led to a large body of theoretical research on failure detectors [TS01, GCG01, ZGSK05, GR06]. Almost all of this work focuses on global failure detection to solve problems like consensus, atomic broadcast, etc. These approaches guarantee the completeness property but are not necessarily good abstractions in a MANET since they are devised for systems which do not contain mobile nodes [Sri06]. In addition, due to the unpredictable mobility of devices, the system may not even be perceived as synchronous at any point in time. The best the programmer can do is to use application-dependent information to determine when a failure should be considered as permanent, e.g., if it lasts longer than a certain time period, if the user had full or restricted access to the service, etc.

Note that we are not arguing against abstractions that enable processes to detect failures. On the contrary, representing failure detection as an abstraction has the advantage that it allows developers to reuse failure handling concerns in other applications [FGF99]. We are only arguing that those abstractions will only be able to detect failures unreliably. To deal with the resulting uncertainty, they should support the description of failures using application semantics.

A high-level representation of failures actually reveals yet another degree of decoupling: it decouples low-level network connectivity of processes from high-level application connectivity. This implies that the aspect of communication is separated from the aspect of failure handling. However, being aware of the state of the connection of a process is still important in a MANET setting because processes must sometimes take explicit action upon connectivity changes. Moreover, those actions often pervade the entire application up to the user interface (e.g., a chat application may grey out participants that unexpectedly disconnect leaving a room). To this end, the failure handling model should also provide abstractions to monitor changes in the underlying network. We state the importance of offering abstractions to monitor connectivity with other processes as our last criteria for communication.

**Criterion 3 (Reacting to Network Connectivity)** A failure handling model should enable processes to monitor the network connectivity of other processes.

[2] A synchronous system assumes that there is a time bounds for computation and communication, while a partially synchronous only makes those assumptions *most of the time* [GR06].

### 2.5.2 Partial Failures and State Consistency

One of main challenges that developers need to face in a MANET setting is to build applications that continuously adapt to a highly dynamic environment and continue working in the presence of partial failures. Failure detection is usually followed by corrective actions whose aim is to bring the system back to a correct state eliminating the detected error. In traditional distributed systems, rollback, rollforward and compensation are the three main techniques applied for system recovery [ALRL04]. In a MANET, such techniques are not suitable:

- Backward recovery techniques (also known as rollback recovery protocols) typically save the state of the system on stable storage at certain points in time (called *checkpoints*), and restore a previous correct state when a failure occurs. However, checkpointing requires distributed snapshots, i.e., a global view of the system. In a MANET environment, processes cannot get information about a global state of the system as processes may experience unexpected partial failures. Moreover, due to the lack of central administration, it is not possible to guarantee that every process is cooperative and is making snapshots.

  Log-based techniques were introduced to reduce the number of checkpoints while enabling recovery by allowing processes to replay their execution after a failure beyond the most recent checkpoint. Despite simplifying failure recovery, such techniques provide transparent support which does not require any intervention of the application or the programmer [EAWJ02]. This makes them unsuitable for MANETs since applications must be aware of the fact that their environment is continuously changing as they may need to adapt their behaviour to those changes, e.g., location changes, or network disconnections.

- Rollforward (also known as forward recovery ) aims to bring the system in a new correct state from which it can continue executing, instead of moving back into a previously correct state. The main problem with rollforward is that it requires prior knowledge of which failures may occur, i.e., it can only move into a correct state when all possible failure situation are known beforehand.

- Finally, compensation techniques aim to incorporate enough redundancy in the system to enable a failure to be masked. Failure masking thus stems from the systematic use of compensation. Examples of those techniques include atomic transactions, replication, group-based abstractions, and migration of objects. In a nutshell, those techniques ensure the availability of services in the network by providing temporal redundancy in which actions may be performed again if needed (as in the case of transactions), or data redundancy in which data is duplicated in other processes so that when a process fails, the data is still available (as in the case of replication, and group-based abstractions). However, they generally do not scale to MANETs because they assume the network is mostly reliable (i.e., nodes are fixed and have a more or less stable network connection) and rely on some centralized coordination authority. Moreover, those techniques are useless if a partial failure isolates a node from all the other ones.

The root problem of recovering from partial failures in a MANET is that a fundamental trade-off needs to be made between consistency and availability of data. Such a trade-off was first remarked in the domain of web services as Brewer's conjecture and

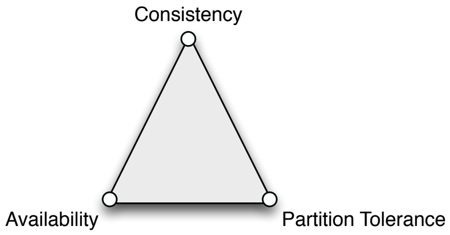


Figure 2.2: The CAP theorem.

later, formalized and proven as the CAP theorem in [GL02]. The theorem, depicted in Figure 2.2, states that it is impossible to simultaneously provide consistency, availability and partition tolerance; only two can be guaranteed at the same time. Since MANET applications must almost always be partition-tolerant because failures are inherent to the network topology, a choice needs to be made between providing consistency or availability. Relaxing consistency allows the system to remain highly available in the face of partial failures, while emphasizing consistency implies that the system may not be available under certain conditions. Since not having access to services can be considered the rule in a MANET, trading off consistency provides a scalable solution. Not only providing consistency usually introduces communication overhead, it may be also difficult (if not impossible) to achieve without making assumptions about the mobility of devices [MP06]. Based on the previous observations, we state the following criterion:

**Criterion 4 (Local Failure Recovery)** Processes should rely as little as possible on remote parties to recover from partial failures. They should be able to perform failure handling based on their local state as much as possible.

Considering the opportunistic nature of communication which may prevent applications from accessing a service and data for extensive periods of time, failure handling abstractions should be based on mechanisms that increase data availability while relaxing consistency. In the literature, *disconnected operation* techniques provide data availability by emulating the functionality of the disconnected service locally. They were first introduced in the area of distributed file systems (in the Coda file system [KS92]), but some techniques have been applied in traditional distributed applications. The most common techniques used for supporting disconnected operation include caching, hoarding (i.e., prefetching the data likely to be used before a disconnection), queueing remote interactions during disconnection, and re-routing remote communications [MRM06]. In short, each process should be able to react to failures without relying on other remote processes.

Enhancing availability of services and data in the face of partial failures is suitable for applications that can handle slightly outdated data. For example, it has been used to build high-volume web services such as Amazon and Google [Vog09]. However, in a mobile setting, the way to react to failures is highly application-specific [Nik00]. We explicitly state the importance of this observation in the following criterion.

**Criterion 5 (Application-dependent Failure Handling Strategies)** A failure handling model should enable processes to define the most appropriate strategy to react to partial failures.

Although disconnected operation mechanisms provide a “default strategy” for dealing with partial failures, developers still have to do part of the work. For some applications, it may be acceptable to wait for the connection to be repaired to resume its task, e.g., distributed video streaming applications. For others, it may be more convenient to continue their task with a substitute service available in the proximity. As such, failure handling abstractions should be flexible enough to incorporate a wide range of failure handling strategies. Not only is the strategy to choose highly application-specific, but it may also depend on the kind of interaction in which application is engaged. For example, reacting to the failure at the user’s home, or when the user is at a conference may require different kind of failure handling strategy. In one case, it may be expected that the user will eventually come back home and resume its tasks, while in the latter, he will only remain for a limited period of time at the conference. We conclude that developers should be able to choose the appropriate technique to distribute resources enabling application to cope with partial failures in a dynamic way.

### 2.5.3 Partial Failures and Memory Management

A partial failure may cause the resources held for disappeared peers never to be freed, leading to an accumulation of outdated information. At first sight, one could solve this problem by making each process responsible of the data it produces, i.e. the developer needs to explicitly free resources when they are no longer needed. However, *explicit memory management* is known to lead to two very common bugs: *incompleteness*, the failure to free all used resources (memory leaks), and *unsoundness*, the premature reclamation of resources (dangling references) [AR98]. While incompleteness can lead to indefinitely accumulation of useless resources, unsoundness can lead to unpredictable behaviour which is difficult to detect and debug since entities may fail long after the event that caused the problem. In sequential programming, these issues are typically solved by introducing a *garbage collector*. Many software platforms for distributed computing adopted this technique and incorporate a *distributed garbage collector*[3].

The main motivation of the research in distributed garbage collection is to provide a *transparent* memory management similar to local garbage collection. They follow one of the two well-known families derived from their local counterpart, namely tracing and reference counting. In a mobile setting, these techniques are too inflexible:

- Distributed tracing combines a global inter-space collector (which takes care of the *mark phase* which recursively traverses the distributed object graph marking unused objects) with independent local garbage collectors (which take care of the *sweep phases* during which unreferenced objects will be reclaimed). Although these techniques can collect all kinds of data structures (i.e., they are complete), they make strong assumptions on the reliability of the network and require the cooperation of *all* nodes to collect garbage. As a result, no progress can be made when a node fails, making them unsuitable for a MANET. Some extensions aim to improve scalability and avoid global synchronization by organizing nodes into groups which cooperate to perform garbage collection [LQP92, RJ98]. However, those techniques can only make progress in the face of failures provided that the failed node is not part of the group.

[3] Since most of the research on distributed garbage collection has been investigated in object-oriented systems [AR98, PS95], the term generally denotes an automatic memory management scheme that reclaims unused objects that are held by another object in a remote process.

- Distributed reference counting, on the other hand, associates each remote object with a counter which is updated on each reference creation or deletion. These techniques are typically more fault-tolerant and scalable, but maintaining counters accurately is costly in terms of communication overhead and effort to ensure causal order of events. Many works have been proposed to enhance fault-tolerance and reduce overhead including indirect reference counting [Piq91], generational reference counting [Gol89], weighted reference counting [Bev87, WI87], network objects [BNOW93], and SSP Chains [SDP92]. A well-known disadvantage of (distributed) reference counting is that it fails to collect cyclic garbage (distributed across devices). To solve this issue, some schemes have been proposed to collect cyclic garbage by using complementary distributed tracing schemes to detect and reclaim cycles [Lin92, JJ92, LQP92, JL93, RJ98, VF05], or by migrating objects so that all cyclic garbage is confined to a single device where it can be collected by the local garbage collector [Bis77, SPG90, ML95]. Yet, they heavily rely on communication between nodes to detect garbage objects.

In essence, distributed garbage collection has to face similar issues as local garbage collection: *soundness* and *completeness*. When a device containing references to remote objects disconnects from the network, either the remote objects have to be kept online until the device reconnects (sacrificing completeness – the device referencing the object may never reconnect, keeping the object from being reclaimed) or it eventually takes the object offline such that it can be reclaimed (sacrificing soundness – the device may reconnect and still refer to the object). The higher rate of partial failures to which applications are exposed in a MANET raises questions on what completeness and soundness mean in this setting. This brings us to our first criterion for a memory management scheme suitable for a MANET.

**Criterion 6 (Relaxing soundness)** Disconnected data should be admitted as *valid* state of the distributed memory management model.

Because traditional distributed garbage collection schemes originate from local garbage collection, compromising soundness is deemed to be out of the question [AR98]. Traditionally, when a network failure occurs, the remote reference becomes unusable and it is considered dangling. However, relaxing soundness makes it possible that applications can deal with intermittent disconnections as remote references remain valid during a network disconnection. Simply put, in a MANET, disconnected data is so common due to the volatile connections phenomenon that it should be treated as a normal state of the system.

In general terms, it is not possible for a distributed garbage collector to determine exactly which data is still in use (alive). As such, all garbage collection schemes use some kind of approximation to *liveness*: an object is still alive if it is still reachable (in tracing-based schemes) or referenced (in reference counting-based schemes). Traditional distributed garbage collection schemes use communication between the different nodes hosting the object graph to compute such an approximation. This is because they assume that the nodes are interconnected by considerably reliable networks such as the internet. In a MANET, network failures arise much more frequently; thus, the problem boils down to knowing when the communication will be restored in order to ascertain if the remote object is still in use. However, as we already argued, the system cannot distinguish a transient from a permanent failure. As a result, traditional distributed garbage collection schemes are not necessarily good techniques in a MANET setting.

Rather than maintaining data until it is no longer used, the lifetime of data should be agreed before engaging in a collaboration such that resources can be freed if a network failure persists. A distributed memory management scheme for a MANET should thus satisfy the following criterion.

**Criterion 7 (Contractual memory management)** The lifetime of data should be agreed *before* being shared among distributed parties.

Note that contractual memory management does not imply an explicit treatment of memory management. We are arguing that transparent memory management *cannot* be reconciled with the characteristics of MANETs: distributed garbage collection requires application-specific information to reclaim garbage. When a datum is first shared, both communicating parties should establish a contract which describes under which circumstances the datum is meaningful in order to help the collector to approximate its liveness. A good failure handling model should enable developers to steer distributed garbage collection while abstracting away as much as possible the many low-level memory management issues.

## 2.6 Partial Failures and the Software Development Process

This dissertation places emphasis on programming language design for easing the systematic construction of applications running on a MANET. To this end, in the previous section, we identified a number of criteria that a failure handling model for building MANET applications should incorporate. Good software development *tools* are also indispensable for promoting the development of applications since they help developers to face better the complexities of software development. In this section, we revise the criteria of a failure handling model in order to support the development of software tools for MANET applications.

The software tools we envision include provisions for dealing with the fact that partial failures are inherent to the application being created, tested, debugged, or maintained. An approach often employed in software development is to incrementally build software focusing on meeting the functional requirements of an application, and considering at a later stage the non-functional requirements. Usually the first increments are the most important requirements because they establish the architectural structure of the system while later increments require, hopefully, only small changes to the overall system. Due to the hardware phenomena inherent to MANETs, failure handling can no longer be considered as a non-functional requirement to be addressed in later increments. Since partial failures may percolate from the underlying middleware up to the graphical user interface of an application, the need arises to support partial failures at the tool level. After all, one can expect that MANET applications incorporate failure handling as a “functional” requirement. This observation led us to investigate tool support in tandem with the programming support for dealing with partial failures.

As previously mentioned, we explore tool support in the form of a debugger. Debuggers assist developers in the process of looking for the places in which the execution of an application deviates from its intended behaviour. In order to detect anomalous behaviors at early development stages, it is important to be able to reproduce a range of possible states that a distributed application can be in, and expose the application to them. To this end, we propose to incorporate means to make software entities fail on purpose in the failure handling model:

**Criterion 8 (Forcing Failures)** A failure handling model should enable processes to cause failures artificially, and trigger failure handling explicitly even if there is no physical network failure.

Forcing failures provides the necessary support for software development tools like debuggers, monitors or testing facilities to define different network topologies. The effects of failures on applications can then be verified by means of the support for reacting to network connectivity (described as criterion 3 in Section 2.5.1). Note that causing the failure of processes also enables applications to trigger failure handling code proactively at specific points in their execution.

Since many bugs may not manifest themselves until the application is actually deployed, software development tools must also be able to operate on deployed mobile systems and obtain information about the network state. To this end, software tools should be able to monitor the network connectivity of processes. This shows once more that the reacting to network connectivity criterion is essential for supporting the development of software tools for MANET applications.

| | |
|---|---|
| *Criteria for Communication* | |
| **C1 Decoupled communication** | Processes should be decoupled as much as possible in space, time, synchronization, and arity to allow computations to proceed in the face of the intermittent failures and zero infrastructure phenomena present in MANETs. |
| **C2 High-level representation of failures** | Processes should agree on criteria to determine when their *logical* communication has terminated. |
| **C3 Reacting to Network Connectivity** | A failure handling model should enable processes to monitor the network connectivity of other processes. |
| *Criteria for State Consistency* | |
| **C4 Local failure recovery** | Processes should rely as little as possible on remote parties to recover from partial failures. They should be able to perform failure handling based on their local state as much as possible. |
| **C5 Application-dependent failure handling strategies** | A failure handling model should enable processes to define the most appropriate strategy to react to partial failures. |
| *Criteria for Memory Management* | |
| **C6 Relaxing soundness** | Disconnected data should be admitted as *valid* state of the distributed memory management model. |
| **C7 Contractual memory management** | The lifetime of data should be agreed *before* being shared among distributed parties. |
| *Criteria for Tool Support* | |
| **C8 Forcing failures** | A failure handling model should enable processes to cause failures artificially, and trigger failure handling explicitly even if there is no physical network failure. |

Table 2.1: Overview of the criteria of a failure handling model for MANETs.

## 2.7 Conclusion

In this chapter, we identified three major design dimensions for the development of a failure handling model suitable for MANETs - communication, state consistency and memory management. We discussed the issues that the high rate of partial failures inherent to MANETs impose on each concern, and derived a set of criteria that a suitable programming model for MANET applications should provide. We also argued that the technological support for each of these dimensions should also facilitate the development of software tools for MANET applications, and subsequently revised our criteria. Figure 2.1 provides an overview of all criteria according to the different concerns which we summarize as follows:

- The issues of communication in MANET are related to the fact that connections are volatile and networks are usually un-administrated. The criterion of decoupled communication (**C1**) derived from the AmOP paradigm allows applications to proceed their collaboration when failures are transient. However, some failures may be permanent, and should be dealt with. A failure handling model should thus provide a high-level representation of failures (**C2**) to allow developers decide when a network failure should be considered as a failure at application level. Nevertheless, it should still enable processes to monitor the network connectivity of other processes as the effects engendered by disconnection often pervade the entire application (**C3**).

- The major issue of bringing the system back to a consistent state once a failure occurred is that developers are confronted by the trade-off between maintaining consistency and providing availability of services. Given a MANET setting, providing mechanisms to perform failure handling which requires as little as possible the intervention of remote parties provides a scalable solution (**C4**). However, how to react to failures is highly application-dependent. Thus, a failure handling model should provide means to enable developers to define the most appropriate strategy to react to partial failures (**C5**).

- The frequent transient failures inherent to a MANET poses the question of how a memory management scheme should deal with soundness and completeness. A model for MANET should relax soundness (**C6**) so that data remains valid during intermittent disconnections, and enable that the lifetime of data is agreed before being shared among other processes (**C7**) so that garbage collection can still happen in the presence of long-lasting disconnections.

- Last but not least, we argued that the effects of partial failures percolate up to the software development tools built for MANET applications. This implies that the failure handling model should also incorporate support to make processes fail on purpose to trigger failure handling explicitly (**C8**).

In the next chapter, we review the state of the art of software engineering technology for MANET in the light of these criteria.

# Part I

# Partial Failures in Mobile Ad hoc Network Applications

# Chapter 3

# Related Work

In the previous chapter, we have distilled a number of programming model criteria to be able to deal with partial failures in mobile ad hoc networks (cf. Table 2.1). In this chapter, we analyse various distributed programming platforms that adhere to one or more of these criteria. As stated in the introduction, our work takes a language-oriented approach based on the design philosophy of a *reflectively extensible kernel language* in which a small kernel language is augmented with a reflective interface to enable the construction of programming abstractions from within the language itself. However, a large body of research in mobile computing has been conducted in the field of middleware. Hence, we survey both distributed programming languages and middleware in the context of our criteria for a failure handling model for MANETs. It is important to remark that we do not pursue completeness in this survey, but we focus on the representative approaches which provide support for one or more of the criteria. In our discussion, we use the acronyms **C#** shown in Table 2.1 to refer to a particular criterion. In general, we will not only focus on approaches developed especially for mobile ad hoc networks since many interesting ideas have been developed for traditional fixed distributed systems.

## 3.1 Survey of Distributed Programming Languages

Programming languages are often considered suitable for expressing distribution and concurrency concerns because they offer higher-level abstractions to deal with the complexities engendered by those concerns [BST89, BGL98, VA01]. Much research in distributed programming languages has focused on general-purpose distributed computing for local area networks (e.g., Emerald [JLHB88], Obliq [Car95] and ABCL/f [YBS86]), wide area networks (e.g., Erlang [AVWW96], Mozart [HRBS98] and Salsa [VA01], or wireless sensor networks (e.g., nesC [GLvB$^+$03], SpatialViews [NKSI05] and ActorNet [KSMA06]). Some other distributed languages have been especially designed for reliable distributed computing (like Argus [Lis88]), secure distributed computing (like E [MTS05]), high-performance computing (e.g. X10 [CGS$^+$05] and Fortress), or web programming (like HOP [SGL06] and Links [CLWY06]).

However, none of these languages provide support to deal with the radically different network topology of MANETs. In general, most languages assume a relatively stable network topology and introduce tight coupling between processes. In particular, they do not feature space-decoupled communication because they assume infrastructure to introduce distributed processes to one another in the network via some form of explicit addressing (e.g., a URL). They do not support either arity decoupling directly since communication happens typically point-to-point. Moreover, the failure handling model is usually aligned with the exception handling model because network failures are considered to be exceptional. Nevertheless, several languages provide interesting features for failure handling, namely Argus, Mozart and E. In the remainder of this section, we further analyze them; we briefly summarize the language's main features before evaluating it. The evaluation is organized according to the three design dimensions of a failure handling model described in Section 2.4.

### 3.1.1 Argus

Argus [Lis88] is a distributed programming language that provides built-in transactional support to cope with the effects of partial failures. The language provides distributed objects (called *guardians*) that support transactional semantics. A guardian is a special kind of remote object which can be accessed by means of *handlers* which are similar to methods. A handler invocation is served by one of the guardian's internal processes. The state of a guardian consists of a set of local objects (which serve handler invocations), and references to other guardian objects. Local objects which are annotated with the `stable` keyword are periodically stored so that they are recoverable after a crash. The rest of the objects are volatile, i.e., they are assumed to contain volatile data that can be discarded or reconstructed from stable objects after a crash. Argus allows computations to run as atomic actions by introducing *atomic* objects. Atomic objects differ from regular ones in the way their operations are handled. Each operation on an atomic object has a lock (either a read or write lock). Operation invocations annotated with the `action` keyword are then handled as a transaction in which state changes are applied to a copy of the state of the atomic object. Argus ensures that either the operation is completed successfully (and state changes are committed), or no changes are made to the object.

**Communication** Argus supports both synchronous and asynchronous remote methods invocations. Asynchronous remote methods invocations are supported by means of *streams calls*. Argus's streams allow a client object to run in parallel with the object processing the invocation, thus, decoupling them in time. Stream calls are buffered and sent conveniently so that multiple messages can be sent without waiting for them to be processed. When a handler invocation is performed on a guardian, Argus delays its execution until all earlier invocations are completed, ensuring that invocations are executed in the correct order. Streams also have some built-in support to deal with failures as the system retries to deliver calls that failed due to a communication problem with the guarantee that they are executed at most once. However, the system only retries to deliver calls for an unspecified period of time after which it gives up and breaks the stream, raising a failure exception.

In order to deal with the return values of asynchronous remote method invocations, Argus employs the notion of futures which are also known as *promises* [LS88]. A promise serves as a proxy for the value which will be returned as result of the asynchronous remote invocation. Argus pioneered the concept of *promise chaining* which allows one to "chain" asynchronous remote invocations which results in immediate sends without having to wait for the method's return values. In order to support synchronization on the result represented by the promise, the language provides the `claim` operation which allows one to “wait” for the result of the asynchronous remote invocation. The operation suspends the calling thread if the value of the promise was not received when it was called. As such, Argus does not feature full synchronization decoupling.

**State consistency** The language aims to provide a transparent solution for dealing with the effects of partial failures by means of atomic actions. As such, it does not provide explicit means of application-dependent failure handling strategies (**C5**). Atomic actions provide an all-or-nothing semantics to deal with failures during remote method invocations. However, atomic actions are based on a two-phase-commit protocol which is not feasible in the context of MANETs. The main problem is that the protocol expects failures to be rare and that the participants will come back online to provide their answer, but in a MANET volatile connections are common and failures may not be resolved within a reasonable amount of time due to mobility of devices.

**Memory management** Argus’ memory management model does not relax soundness (**C6**) nor supports contractual memory management (**C7**). It provides a transparent memory management in which each node incorporates a stop the world, mark and sweep, compacting garbage collector [LCJS87]. In [LL86], Liskov et Ladin describe a distributed garbage collection algorithm which relies on a local garbage collector for reclaiming local objects, and introduces a centralized service to store information about inter-node references. The idea is that each node periodically performs local garbage collection independently, and updates the information about their references to remote objects. Although the algorithm is designed to tolerate network failures, it assumes that “nodes do eventually recover from crashes” [LL86].

### 3.1.2 Mozart

The Mozart Programming System [HRBS98] is the primary implementation of the OZ language, a multi-paradigm language that supports functional programming, object-oriented programming and constraint logic programming. It was implemented within the context of the Distributed Oz project whose goal was to provide a distributed model for OZ built around the principle of separation of concerns [Roy99]. Mozart adheres to the classic goal in traditional distributed systems of *network transparency*, i.e., a program should run with the same semantics independently of the distribution of its language entities. As long as there are no communication issues, remote or local entities are indistinguishable to programmers. Yet, the distribution model is also *network-aware*, i.e., a program has some control over the network connectivity of entities.

A distributed application in Mozart consists of a number of *sites* across the network hosting language entities (e.g., objects, dataflow variables, etc.). Sites communicate by means of shared entities, which can be of three different sorts: mutable entities (including cells, ports, objects, threads and locks), monolithic entities (mutable entities which can be only assigned once, including streams and dataflow logic variables), and immutable entities (including simple values such as numbers, records, and procedures). Each type of shared entity is attached to a distributed protocol which determines its network behaviour based on the value to some distributed parameters. There are three orthogonal distribution parameters which can be attached to an entity: (1) the *access architecture* parameter which defines how the sites sharing an entity are coordinated, (2) the *state consistency* parameter which defines where the state of the entity is located, and (3) the *reference consistency* parameter which defines the distributed garbage collection algorithm used on the entity. Although each entity carries a default annotation, programmers can alter the default distributed protocol by means of `Annotate` procedure which applies a list of distribution parameters for a given entity.

Mozart provides a failure handling model which maps the failure detection mechanism in the network layer onto failures at the programming level. A language entity can be either OK, temporarily failed, or permanently failed. A program can detect the failure of an entity using two different mechanisms: watchers and handlers. A watcher allows developers to attach a procedure for a given entity which is invoked when the entity enters the failed state that the watcher is configured for. A handler attaches a procedure for a given entity with certain conditions of activation, but the handler procedure is only called when attempting an operation on a failed entity. The common type of handler raises an exception into the calling thread, but programmers can provide a handler procedure that replaces the attempted operation. In short, watchers perform eager failure detection (as they trigger failure handling code when the entity fails even when no explicit operation is performed), while handlers perform failure detection lazily (as they only trigger upon operations on entities). Collet remarked in [CR06] that watchers also provide asynchronous failure handling since the watcher procedure is invoked in its own thread, while handlers are the synchronous counterparts since the handler procedure is invoked in the thread attempting an operation on a failed entity.

In his dissertation Collet [CR06] revisits Mozart's failure model and proposes a fully asynchronous failure handling model based on *fault streams* instead. In his model, every site has a failure detector for each language entity which produces a stream giving its failure state transitions. He distinguishes between an entity that is permanently failed from the viewpoint of one site (i.e., other sites may still have access to the entity), or globally for all sites. As a result, a language entity can be either OK, temporarily failed, locally failed, or permanently failed. Programmers can monitor the state of an entity by reading its fault stream. A fault stream of an entity thus reifies the history of states of that entity. This means that programs can now react by a transition back to the state OK which was not possible in the previous failure model.

**Communication** Mozart does not decouple entities in space because the language assumes infrastructure to introduce language entities to other sites in the network. In order for a site to acquire an initial reference to a shared entity, it has first to acquire (e.g., by mail, web page, etc.) a string representing the entity's global identity. Second, synchronization decoupling is not fully supported because the language allows a thread to suspend on certain circumstances. Quoting Collet in his dissertation [Col07] "an operation on a failed entity simply blocks until the entity's fault state becomes OK again.". The operation may resume if the failure is temporary, but it suspends forever if the failure is permanent. This also conflicts with the contractual memory management criterion (**C7**) since the blocked thread keeps the entity alive, preventing the memory management system from reclaiming it. To solve this issue, programmers explicitly need to make an entity fail. When a fault stream is no longer alive, the system closes the stream (i.e., it triggers the state **`nil`**). The threads monitoring the entity can then detect this state and manually remove explicit references to the entity. Finally, arity decoupling is not directly supported in the language because references to shared entities are point-to-point.

Although fault streams allow programmers to react to changes of network connectivity (**C3**), they do not directly support a high-level description of failures (**C2**). The transitions from temporary to permanent are determined transparently by the underlying failure detection mechanism, which is in turn, built on top the network failures of TCP. More high-level representations of failures need to be encoded manually by means of fault streams and handler/watchers procedures.

**State consistency** Mozart's distributed parameters provide a form of application-dependent failure handling strategies (**C5**) since developers can select the most appropriate built-in strategy to deal with data inconsistencies and memory management. However, the `Annotate` procedure can only be called *before* the entity gets distributed. Once an entity is distributed for the first time, its distribution parameters can no longer be changed. This may be too restrictive for certain MANET applications which need to dynamically adapt their behaviour to changes in the environment.

**Memory management** In Mozart, developers can choose among three built-in strategies in the reference consistency parameter: `persistent` (in which the system simply keeps the entity alive forever), `refcount` (in which the system reclaims the entity based on a weighted reference counting algorithm [Bev87, WI87]), and `lease` (in which the system reclaims the entity after its lease expires). Choosing a lease strategy for an entity allows for relaxing soundness (**C6**) and contractual memory management (**C7**). Although developers can choose a memory management strategy, the language still provides transparent memory management. As such, to the best of our knowledge, developers cannot choose the lease duration for a given entity, nor deviate from the default lease strategy behaviour (e.g., renew leases after certain conditions are met).

We conclude Mozart's evaluation by remarking that Collet introduced in [Col07] support to force the failure of an entity (**C8**) either locally or globally.

### 3.1.3 E

E [MTS05] is a distributed object-oriented language which was designed for secure peer-to-peer distributed programming for open networks such as the internet. The language combines actors and objects into a unified concurrency model called the *communicating event loops model*. The model is an adaptation of the actor model [Agh86] in which actors are represented as *vats* (containers) of regular objects, rather than an active object (as done in previous actor languages such as ABCL). A vat consists of a heap of objects and a thread of control which perpetually processes messages from the actor's message queue and invokes the corresponding method of the object. Each object is said to be owned by exactly one vat, and a vat may host multiple objects. Each object can be referenced from objects owned by other vats. Rather than representing local and remote objects differently, E distinguishes between several kinds of references which define the kind of invocations supported. Objects within the same vat refer to one another by means of *near references* which can carry synchronous or asynchronous method invocations. Objects hosted in different vats can only communicate using asynchronous method invocations by means of *eventual references*.

**Communication** Since eventual references are the only kind of object reference that can span across different devices, all distributed communication in E is asynchronous by design. This makes the language quite suitable for a MANET. However, E's eventual references are not designed to express communication over volatile connections.

A network failure immediately breaks any vat-crossing eventual reference (a remote promise or a far reference). This implies that any message sent after the connection is lost, and the message's promise is resolved with an exception. As such, E's event loop concurrency model does not feature decoupling in time.

In order to deal with the results of asynchronous method invocations, E also employs promises inspired by Argus's promises. However, E pioneered the `when` construct which allows one to access the value of a promise in an entirely non-blocking, event-driven manner. As such, in contrast to Argus, E's event loop concurrency model features full synchronization decoupling.

E introduces support for partial failure handling that allows developers to explicitly manage the disconnection of a reference. E's references respond to a method called `whenBroken` that takes as argument a handler that is registered to be notified when the reference breaks upon a network partition. Interestingly, references also respond to a method called `reactToLostClient` that notifies the target objects that at least one of its clients may no longer be able to reach it. However, the language does not fully feature support for reacting to network connectivity (**C3**), since there is no corresponding method for reacting to the reconnection of a reference. This is because, quoting Miller in his thesis [Mil06], "even after a partition heals, all references broken by that partition stay broken". In order to regain connectivity to a remote object after a network failure, E introduces the notion of *sturdy references*. In contrast to traditional object references, sturdy references do not carry messages from a client to a target object. Rather, a sturdy reference allows one to ask for a new "live" reference to the target object when an existing one breaks upon a network partition. A sturdy reference is a form of *offline capability* which contains the information needed to authorize access to a given target object. Sturdy references, however, are created by means of an explicit address (in the form of a URI string), so they do not decouple objects in space.

**State consistency** The `whenBroken` and `reactToLostClient` methods allow developers to schedule correcting actions at both ends of the reference when a failure occurs. These methods thus provide the basis on which to build application-dependent failure handling strategies (**C5**).

**Memory management** To the best of our knowledge, E features transparent memory management based on a distributed garbage collection algorithm derived from reference counting. Miller mentions in [MTS05, Mil06] that the algorithm assumes that each vat determines when a reference is no longer reachable, and it does not collect unreachable inter-vat reference cycles. Since reference breaks upon a network partition, E's memory management model does not relax soundness (**C6**). However, some control is provided over the lifetime of offline capabilities encapsulated by a sturdy reference. When a host creates a sturdy reference for one of its local objects, it may associate a *future date* with the reference, denoting how long the host should keep the object available. As such, the language features contractual memory management (**C7**) only for sturdy references.

## 3.2 Survey of Formal Languages for Distributed Computing

A large body of research in distributed computing has focused on designing formal models and languages rather than concrete programming languages for describing distributed computations in open networks. Many approaches are based on the actor model [Agh86] (e.g., the ActorSpace model [AC93]). In the actor model, a system consists of a number of concurrent, autonomous entities called *actors* which communicate with one another by sending asynchronous messages. Each actor has a mail address (uniquely identifying an actor), a message queue (a mailbox storing incoming messages) and a behaviour (defining how an actor handles incoming messages). Since an actor's communication model decouples actors in time and in synhronization, the model is of great relevance in the field of mobile computing. Some approaches aim to reconcile the asynchronous communication model from actors with the conventional object-oriented paradigm such as Creol [JO07].

Another group of models and languages are based on the concept of *coordination*. A coordination model is concerned with the communication between different entities in a system, i.e., "the glue that binds separate activities into an ensemble" [PA98]. It can be represented by a <E,L,M> tuple in which E denotes the entities being coordinated, L the mechanism used to coordinate them, and M the semantic framework the model complies with. A coordination language embodies a coordination model, i.e., it implements a communication model rather than a computational one. Most coordination models and languages for concurrent and distributed systems are based on the tuple space model of Linda [Gel85]. Since much mobile computing middleware is derived from the tuple space model, we discuss it in a separate section later (cf. Section 3.3.2).

Although many formal languages provide fully decoupled communication, they do not provide explicit means for failure handling. A notable exception is the failure handling model proposed in [JLZ11] for the Abstract Behavioural Specification language (ABS) [JHS$^+$10]. ABS is a concurrent object-oriented modeling language of which the concurrency model is based on the concurrency model of Creol. The ABS communication model features asynchronous method calls with futures as return values. The language provides both blocking and non-blocking constructs to synchronise active objects on return values by means of the `get` and `await` primitives, respectively.

The failure handling primitives introduced in [JLZ11] to ABS's communication model employ futures both to notify failures, and to coordinate error recovery between client and receiver objects of an asynchronous method call. More specifically, they introduce the `abort` primitive which takes as argument a *fault name* and notifies the client object of an asynchronous method call (the caller) about the failure of the receiver object. Client objects can detect whether the invocation failed using an extended version of a `get` primitive which takes as argument a fault name and a clause to execute if the future contains the given fault name as result. While such a primitive allows processes to detect and deal with failures even if there is no physical network failure (**C3**), failure handling is still represented as an exception. In addition, the model assumes that failures behave in a fail-stop fashion (cf. Section 2.3) but in a MANET, processes do not typically fail gracefully notifying other processes before stopping their computation.

Interestingly, the model introduces a compensating handling mechanism for cancelling out past asynchronous method calls. To this end, they augment ABS's `return` primitive with an `on compensate` clause which takes as argument the compensation code that needs to be executed after the method's normal execution completed if compensation of this call is needed. They also introduce the `kill` primitive which allows to "annul" an asynchronous message call, i.e., it cancels the asynchronous method execution (if it did not start yet) or, it executes the compensation code (if the call was already executed and it successfully completed). The primitive returns a future itself to be used to obtain the result using the regular `await` and `get` primitives. Two special fault names may be returned in such a future that specify either that the method call was successfully annulled before starting or at its end, or that the killed method did not define a compensation. Although compensating handling code is only triggered by a `kill` request at the caller side, it provides an interesting form of application-dependent failure handling strategies (**C5**).

## 3.3 Survey of Mobile Computing Middleware

As previously mentioned, there has been a lot of active research with respect to mobile computing middleware [MLE02, BC06]. The bulk of this research can be categorized into several classes (i.e., object-oriented middleware, tuple space-based middleware, publish/subscribe middleware, and reflective middleware) which we discuss in detail in this section. We also include some interesting middleware platforms that have been designed for nomadic networks (networks composed of a mix of mobile devices and a core infrastructure with fixed nodes) rather than for pure ad hoc networks.

### 3.3.1 Object-oriented Middleware

Traditionally, object-oriented middleware supports distributed communication based on the abstraction of a remote procedure call (RPC) [BN84] in which the client object issuing a method invocation is blocked until the receiver object has returned the result of the computation. Although RPC scales very poorly for a mobile setting [MLE02], several extensions have been proposed to adapt RPC to nomadic networks by supporting queuing of RPCs (e.g., Rover [JdT$^+$95]), or enabling rebinding of resources (e.g., Mobile DCE [SBBK95]). These approaches provide a solution to deal with the effect of volatile connections but, they do not address long-lasting disconnections as they do not provide a high-level representation of failures (**C2**). In the remainder of this section, we highlight the most relevant object-oriented middleware with respect to the criteria we distilled for a failure handling model for MANETs (cf. Table 2.1).

#### 3.3.1.1 Java Intelligent Network Infrastructure (JINI)

Jini [Wal99] is a middleware for service-oriented computing built on top of Java. Its main goal is to allow clients and services to discover and set up an ad hoc network in a flexible, easy manner with minimal administrative infrastructure. The most important concept within the Jini architecture is the service. A service is an entity available on the network which performs a number of tasks for a client. The lookup service is a central abstraction in Jini since it allows services to advertise themselves in the network, and clients to find services by launching queries in the Jini lookup service. In contrast to traditional lookup services, clients and services do not need to be configured with the network address of the lookup service but they can automatically discover it in the network. The lookup service thus acts as a shared space for offering and finding services in a network (often referred to as a *federation*). Jini thus features space decoupling.

Once services have been introduced, Jini relies on the synchronous communication model of Java RMI [Wal01]. This implies that communication is not decoupled in time nor synchronization as a network disconnection blocks the communication channel between objects. Nevertheless, Jini's architecture is flexible enough to support time, synchronization and arity-decoupled communication. Jini distinguishes between the Java interface of the object to be advertised (which must be known by the client object) and the implementation of a proxy object supporting the same interface. Once a client has downloaded a service, the proxy is the communication channel to the service. As such, in theory, the proxy can implement different communication models to interact with its service [Wal01]. In practice, to the best of our knowledge, Jini does not offer any implementation that deviates from the Java RMI semantical model based on remote object references which carry synchronous method invocations.

Jini employs the concept of leasing to allow devices leave the network gracefully without affecting the rest of the system. Quoting Waldo in [Wal01], "the Jini system encourages the establishment of such relationships for a finite duration through the granting of leases on the relationship". When services register themselves with the lookup service, they obtain a lease (of which duration is determined by the lookup service itself). Services must then renew their lease with the lookup service in order to keep their registration in the lookup service. If they cannot, the lookup service removes the service registration when the lease expires. The lease expiration thus denotes that the relationship covered by the lease is ended. Services can also explicitly cancel the lease prior to its expiration. It is important to remark lease expiration is not causally connected with the underlying state of the network connection. Hence, leasing provides a high-level representation of failures (**C2**) since it allows services to agree on a criteria to determine the validity of their relationship with the lookup service.

Leasing was originally brought to Jini to deal with the effects of partial failures and simplify resource management. If the lease covered the allocation of some resource, the resource associated with the lease can be cleaned up after the lease expires. The Jini specification advocates a use of leasing to mediate the relationships to any kind of resource including access to objects (references), files, event registrations, certificates that grant the lease holder certain capabilities, or service registrations [jin03]. However, to the best of our knowledge, leases have only been integrated in the Jini lookup service (as previously explained) and JavaSpaces [FAH99]. JavaSpaces is a Jini service providing an implementation of the original tuple space model of Linda (see also Section 3.3.2). In JavaSpaces, tuples are modeled as Java objects and can be inserted in the tuple space specifying a lease time, which specifies the maximum amount of time for which they can reside in the tuple space before being automatically removed. To sum up, Jini offers "contractual resource management" (**C7**) as leasing allows entities to agree on a time-based convention when they establish a relationship. However, due to its reliance on Java RMI, Jini's memory management model does not relax soundness (**C6**) since a remote reference becomes broken upon a network disconnection.

We will revisit Jini in Section 6.10 where we describe the differences and similarities between its leasing model and the one proposed in this work.

#### 3.3.1.2 Rover

Rover [JdT$^{+}$95] is a software toolkit designed to ease the construction of both *mobile-transparent* and *mobile-aware* applications. Mobile-transparent applications aim to hide the effects of mobility to the application, while mobile-aware applications are aware of mobility such that they can react appropriately. Rover assumes a nomadic network in which clients run on mobile devices, and servers run on stationary devices. Communication happens solely on a client/server basis and as such, mobile devices do not directly communicate with one another. While this makes Rover look not very suitable for ad hoc networks, it offers some interesting mechanisms for failure handling.

Rover supports local failure recovery (**C4**) by incorporating support for disconnected operation. To this end, it introduces two abstractions: relocatable dynamic objects (RDOs) and queued remote procedure call (QRPC). RDOs behave as mobile objects which can be dynamically migrated to a client from a server or vice versa. However, a RDO always resides on a server which maintains the primary copy. Clients can obtain secondary copies enabling them to work with their local copy during disconnections. Updates on that objects need to be reconciled with the primary copy when connection between client and server is available. Conflicts resulting from concurrent modifications can be handled in an application-dependent way since one can provide a conflict resolution strategy for each object. QRPC, on the other hand, provides an extension to RPC model in which RPCs performed during a disconnection are buffered in a stable log. When the connection between client and server is available, they are redelivered and deleted from the log when a result has been received from the server. Clients can either block while the QRPC is pending or register a callback which will be asynchronously invoked upon the arrival of the result (providing thus a form of synchronization decoupling).

By means of RDOs and QRPC, Rover offers two built-in failure handling strategies (**C5**). The applicability of RDOs is, however, limited in a MANET because traditional replication techniques do not scale to MANETs [Ded06]. QRPC, on the other hand, does provide time-decoupled communication between client and server. Finally, Rover has built-in support to allow applications to react to changes in their *environment*. The environment consists of the state of the RDOs, and the state of the network. As such, Rover enables processes to monitor the network connectivity of other processes (**C3**). Applications can either poll or register a callback to determine the state of the environment.

#### 3.3.1.3 DR-OSGi

DR-OSGi [KTA09] is a distributed service-oriented middleware designed to deal with the effects of volatile connections. It extends the R-OSGi distributed infrastructure for OSGi components with *hardening strategies* to enable applications to continue working when the underlying network becomes unavailable. DR-OSGi goes one step further than Rover on support for disconnected operation. It allows applications to choose among a catalog of disconnection operation techniques derived from Mikic-Rakic and Medvidovic survey [MRM06]. The authors describe five techniques: caching (locally storing a subset of remote data that has been accessed), hoarding (prefetching the likely needed remote data before a disconnection), queueing (buffering remote requests similar to QRPC), replication (maintaining a local copy similarly to RDOs) and multi-model components (which allows to combine several of the prior strategies). However, to the best of our knowledge, DR-OSGi only provides cache, queueing and replication by default.

Interestingly, DR-OSGi has been designed as an extensible framework in which developers can implement custom hardening strategies. It augments the R-OSGi infrastructure with a hardening manager and a collection of hardening strategies (which

Listing 3.1: The DisconnectionListener interface.

```java
public interface DisconnectionListener{
  public Object disconnectedInvoke(RemoteCallMessage invokeMessage);
  public Object reconnected( String uri );
  public void remoteInvoke(RemoteCallMessage invokeMessage, Object result);
  public void serviceAdded(String uri);
  public void serviceRemoved(String uri);
}
```

can be configured by the programmer). The hardening manager intercepts the handling of network exception and successful reconnections attempts from R-OSGi by means of aspect oriented programming technology. In response to those events, it starts and stop the corresponding hardening strategy. More concretely, programmers have to provide a configuration file that specifies which hardening strategy should be applied to which application. More than one strategy can be specified, which are applied in the order defined in the configuration file. If the first strategy succeeds, DR-OSGi does not apply the second one.

In order to create a new hardening strategy, the `DisconnectionListener` interface (shown in Listing 3.1) needs to be implemented. The `disconnectedInvoke` and `reconnected` methods allow programmers to intercept the disconnection and reconnection procedures of R-OSGi. Similarly to Mozart fault streams, these methods allow the programmer to react to network connectivity (**C3**), but they do not directly support a high-level representation of failures (**C2**). The `remoteInvoke` method is called when the remote service invocation succeeds. The implemented class then has to be deployed as a regular OSGi bundle.

DR-OSGi provides essentially application-dependent failure handling strategies (**C5**). Furthermore, failure handling strategies are expressed as separate components, promoting their reusability among different applications. However, since a hardening strategy is applied to all services within an application, this may be too coarse grained for certain applications. In addition, the fact that the `remoteInvoke` method takes as argument the result of the invocation indicates that DR-OSGi relies on a synchronous communication model. Hence, it does not support decoupled communication (**C1**).

### 3.3.2 Tuple Space-based Middleware

Tuple spaces were first introduced in the coordination language Linda [Gel85]. A tuple space is a globally shared virtual data structure which allows processes to communicate by posting and reading *tuples*. A tuple is an ordered group of values (called the *tuple content*) and has an identifier (called the *type name*). Processes can post and read tuples using three basic operations: `out` to insert a tuple into the tuple space, `in` to remove a tuple from the tuple space and `rd` to access a tuple present in the tuple space (without removing it). Tuples are anonymous and are extracted from the tuple space by means of pattern matching on the tuple content.

Tuple space communication is decoupled both in space and time: processes do not have to know each other beforehand nor be online at the same time in order to insert and extract tuples. These characteristics make tuple spaces a suitable communication model for a mobile setting [MLE02]. However, maintaining a globally shared tuple space is not compatible with the hardware characteristics of MANETs (cf. Section 2.1).

Adaptations of tuple spaces designed for mobile computing applications have been proposed that solve this issue. We now review the two most prominent tuple space-based middleware platforms, namely LIME [MPR01] and TOTA [MZ04]. We will revisit tuple space-based middleware in Section 7.9 when discussing work related to the tuple space model proposed in this work.

#### 3.3.2.1 Linda in a Mobile Environment (LIME)

LIME [PMR99, MPR01] adapts the original tuple space model for mobile computing through the notion of *federated tuple spaces*. In this model, mobile *agents* are equipped with a local tuple space called *interface tuple space* (ITS). Whenever devices come into communication range, the ITS are conceptually merged into a federated tuple space which is transiently shared amongst devices. The federated tuple space enables agents to access tuples from remote agents. Since tuples can only be exchanged when the communication partner that issued a tuple space operation is in range of the device that emitted the requested tuple, communication in LIME is by default not decoupled in time. Time-decoupled communication is traded in for guaranteeing atomicity for remove operations which is an essential feature to support synchronization between applications.

In order to allow a tuple to be accessible to other tuple spaces which are not connected to the emitter of the tuple, LIME exploits the notion of *location*. A tuple can be inserted into the ITS with an explicit tuple space location in which the tuple should be placed. If the destination location is currently not connected, the tuple remains at the current location, awaiting the arrival of the destination tuple space. Tuples whose current location is different from the intended destination location are called *misplaced tuples*. Misplaced tuples, however, abandon anonymous, space-decoupled communication so characteristic of tuple space models.

Tuple spaces traditionally do not feature synchronization decoupling because the operations to extract tuples from the tuple space are blocking. To solve this issue, LIME extends the original tuple space model with the notion of *reactions*. A reaction consists of a pattern and a callback that specifies the actions to be executed when a tuple matching the given pattern is available in the tuple space.

LIME enables processes to monitor the connection status of the available agents in the underlying network configuration (**C3**) by means of a special read-only, system-maintained tuple space. More specifically, its tuples provide information the tuple spaces present in the system and on which host they reside on.

#### 3.3.2.2 Tuples on the Air (TOTA)

TOTA [MZ04, MZ09] takes a different approach to distribute the tuple space over the network by adopting a *replication-based model*. Rather than merging local tuple spaces when devices are connected as in LIME, TOTA replicates tuples among collocated devices. TOTA tuples are equipped with a *propagation rule* that determines how a tuple migrates from one tuple space to another. The propagation rule takes the form of a number of operations defined on a tuple which are called by the system when the tuple arrives to a tuple space. They are shown in Listing 3.2. Exchanging tuples according to propagation rules allows the definition of application-specific coordination in which tuples are shared based not only on network connectivity, but also on semantic information. The propagation rule can also change the tuple information (by means

Listing 3.2: The TOTA propagation rule.

```
if(decideEnter()) {
  boolean prop = decidePropagate();
  changeTupleContent();
  this.makeSubscriptions();
  tota.store(this);
  if(prop) tota.move(this);
}
```

of the `changeTupleContent` method), thus providing programmers with a flexible mechanism to achieve context-awareness in an adaptive way.

While TOTA's replication-based model provides decoupling in time, it does not guarantee atomicity for remove operations. In fact, TOTA does not provide built-in mechanisms to remove the whole tuple structure, i.e., a tuple and all its replicas that migrated to other devices in the network. The provided `delete` primitive just extracts from the local TOTA middleware all tuples matching a given template. In [MZ09], besides a propagation rule, the authors augment tuples with a *maintenance rule* which allows tuples to update the tuple field upon network topology changes (so that the original propagation rule is preserved). In this version of TOTA, the `delete` primitive may have an effect on the whole tuple structure depending on its maintenance rule and the invoking agent. If the tuple has a maintenance rule that specifies that its structure should be preserved upon network reconfiguration, then deleting it "from the source node induces a recursive deletion of the whole tuple structure from the network". This allows the agent that emitted a tuple to "unsend" it in the network, but it does not allow other agents to remove tuples as defined in the original model by the `in` operation.

Like LIME, TOTA augments the tuple space model with primitives to notify agents when certain tuples arrive in their tuple space. Connection and disconnections of peers are also represented as tuples, enabling agents to react to network connectivity of the TOTA network (**C3**). However, it is not clear how disconnections are detected in TOTA. In [MZ09], it is described that each TOTA node broadcasts in its one-hop neighbourhood a `PresenceTuple` tuple to announce its presence and that other agents can subscribe to the insertion or removal of these tuples to monitor the network connectivity of an agent. However, upon a disconnection, the source node is not able to notify other agents about the deletion of its `PresenceTuple` tuple.

It is important to note that even though TOTA does not directly offer a high-level representation of failures (**C2**), this can be encoded by exploiting the propagation rules. In the extended version of TOTA described in [MZ09], a `MessageTuple` class defines a tuple that floods the network and deletes itself after some time has passed, thus providing a form of leasing for tuples.

### 3.3.3 Publish/Subscribe Middleware

Publish/subscribe is a communication paradigm in which processes interact by publishing event notification (often called *events*) and subscribing to the type of events they are interested in. Different strategies for specifying event subscriptions are possible depending on the system. For example, in topic-based publish/subscribe, events are matched based on topics or event types, while in content-based publish/subscribe, matching happens on the actual content of the event. The model relies on an *event* *notification service* (or *event broker*) for collecting subscriptions and efficient delivery of events to subscribers.

Many researchers have proposed publish/subscribe as a suitable communication model for MANETs because of its loosely coupled nature [Mei02, CJ02, EFGA03, HGM04]. Not only does publish/subscribe communication decouple publishers and subscribers in time, space and synchronization [EFGA03], but it also naturally supports arity decoupling as events may be delivered to an undetermined number of subscribers. However, the first publish/subscribe systems assumed that components comprising an application are stationary and interact by means of a fixed, reliable network of event brokers which is not a very scalable solution for a mobile setting. Many adaptations for mobile computing relax this assumption by allowing components to be mobile, but they still rely on fixed infrastructure acting as access points to which mobile clients connect from time to time (e.g., SIENA [CRW00], JEDI [CDNF01] and LPS [EGH05]). Publish/subscribe middleware designed for pure ad hoc networks does not rely on intermediate infrastructure and typically employs some form of broadcasting. For example, EMMA [MMH05] broadcasts subscriptions to reachable hosts, while STEAM [MC02] broadcasts events to subscribers within a certain area surrounding the producer.

Publish/subscribe middleware, however, provides no explicit failure handling mechanisms as they typically do no offer abstractions for representing failures (**C2**) or reacting to network connectivity (**C3**). In fact, communicating parties are usually not aware of the underlying network configuration, or even if a published event was received by any subscriber; either failures are transparent to publishers and subscribers and the event notification service buffers events when subscribers disconnect (like in JEDI), or the publishers are responsible themselves for encoding failure handling and events get lost if subscribers move out of communication range (like in STEAM). Two notable exceptions are EMMA and one.world [GDL$^+$04] which we discuss in what follows.

#### 3.3.3.1 Epidemic Messaging Middleware for Ad hoc Networks (EMMA)

EMMA [MMH05] is a publish/subscribe middleware based on the Java Message Service (JMS) [MHS02] in which events are represented by Java objects, and communication between publishers and subscribers happens by means of message queues. Message queues in EMMA, like in JMS, support both point-to-point and (topic-based) publish/subscribe communication. EMMA periodically advertises message queues to nearby hosts, allowing them to automatically discover one another in an ad hoc network. When hosts receive advertisement messages from queues they are subscribed to, they add an entry to their Java Naming and Directory Interface (JNDI) [jnd03] registry. Similarly to Jini, entries in the JNDI are associated with a lease representing the time of validity of a particular entry. If the lease is not renewed, it will eventually expire and the entry is deleted from the registry, i.e., the hosts needs to discover again the queue.

EMMA adapts the concepts of *durable* and *non-durable* subscriptions present in JMS for a MANET setting. A durable subscription remains valid upon disconnection of the clients. On the other hand, a non-durable subscription is deleted upon disconnection, and another subscription needs to be made upon reconnection. Interestingly, if a subscriber is disconnected and the subscription is durable, events are not buffered but are sent using an *asynchronous epidemic routing protocol*. The idea of the protocol is that a message that needs to be sent is replicated to reachable hosts, which in turn send them to all hosts in their range. As such, messages are spread in the network like an infection. While using such an epidemic approach does not guarantee message delivery, the protocol does take care of removing duplicate messages to provide *at-most-*

*once* delivery semantics for messages. Within epidemic routing, each host maintains a buffer storing the messages that it has created and the replicas received from other hosts. Messages are deleted from the buffers using expiration time values that can be set by senders, thus supporting contractual memory management (**C7**).

#### 3.3.3.2 One.world

One.world [GDL$^+$04] is a system architecture for pervasive computing developed on top of Java. It aims to provide an integrated execution platform including a set of system services (e.g., service discovery, migration, remote events, and checkpointing) to ease the development of pervasive computing applications. Applications in one.world consist of an hierarchy of *environments* which can be best compared to processes in operating systems. Environments host application components and the application's persistent data that is encoded by self-describing records with named fields called *tuples*. Application components communicate solely by means of asynchronous event notifications which are represented as tuples. Moreover, one.world supports migration of components between different environments.

Interestingly, one.world supports different communication patterns for event notifications along three design axes. A first point of variation is the *binding time* and determines when to perform a discovery query. Early binding implies that the application first discovers a subscriber, and then uses point-to-point communication for delivering events. Late binding, on the other hand, combines an event notification with discovery into a single operation: an event is routed towards any matching subscriber and successive events may be routed to different subscribers. A second point of variation is the *specificity* and determines the number of subscribers for an event; events are either sent to all matching subscribers or to only a single one. Finally, a third choice is the *query target* that determines the entity on which to perform a discovery query. One may register a discovery query on the external representation of a service (a resource descriptor), or on events themselves.

Similarly to Jini, one.world employes leases for resource management in order to limit the time that resources can be accessed. More concretely, leases delimit the lifetime of tuples in the tuple storage, and "network endpoints" (which interconnect components by means of sockets). Before their expiration, applications can renew these leases to expand their lifespan, or cancel them to relinquish access to the resource. Leases can also be cancelled by the one.world kernel when an application component migrates. What is remarkable is that one-world features some mechanisms for easing the use of leases. The underlying implementation makes use of a *lease maintainer* which automatically renews the lease it controls until the lease is explicitly cancelled. Unfortunately, a lease maintainer is completely hidden from applications which cannot influence its renewal strategy. It is interesting to remark that Grimm et al. conclude in [GDL$^+$04] that although leases are suitable for controlling remote resources, they do not work well for controlling local resources (i.e., tuple storage).

To the best of our knowledge, applications cannot react to the expiration of a lease. As such leases in one.world are used as a form of contractual management (**C7**) rather than providing high-level representation of failures (**C2**). In order to deal with the effects of partial failures, one.world employs checkpointing and migration instead. The system provides a checkpointing operation that saves the state of the environment tree as a tuple, and a restore operation to read a previously captured state and restore its execution. The migration of an application either moves the application's environment and all its content or creates a copy on the remote device. Checkpoints can also migrate to different devices. In contrast to traditional recovery solutions, checkpointing and migration need to be explicitly triggered by applications. In addition, applications are notified after they have been restored from a checkpoint or have been migrated to another device. This allows applications to deal with changes on the execution context. However, such mechanisms still assume that failures can be anticipated and that nodes have time to migrate or checkpoint before the failure occurs, which is no longer the case in a mobile setting.

### 3.3.4 Reflective Middleware

The principle of reflection has been investigated in the field of middleware to offer flexibility for dealing with the highly dynamic nature of the mobile environment [MLE02, BC06]. The main goal of reflective middleware is to allow applications to monitor and adapt the behaviour of the underlying middleware implementation according to its need in order to achieve more efficient or suitable solutions. The first implementations of reflective middleware (including OpenCorba [Led99], OpenORB [BCA+01] and DynamicTAO [KRL+00]) were based on CORBA which follows a RPC model of communication, and as such, they do not support decoupled communication (**C1**). The role of reflection in these approaches had to do with supporting dynamic reconfiguration of the middleware services (e.g., monitoring and security). Reflective middleware designed for mobile computing, on the other hand, employs reflection to offer context-awareness and dynamic adaptation of the middleware behaviour to environmental changes including device and middleware heterogeneity (e.g., ReMMoC [GBS03]), and changes on the execution context and resource fluctuations (e.g., CARISMA [CEM03] and MobiPADS [CC03]). However, some of these systems (such as UIC [RKC01] and ExORB [RI04]) still offer by default communication based on standard object-oriented RPC platforms like CORBA or JavaRMI. Although in principle nothing prevents developers from implementing custom communication components supporting decoupled communication, Grace and Blair argue in [BC06] that ORB-based middleware is too limited to tackle the variety of communication models used in a mobile environment such as publish/subscribe, tuple spaces and data-sharing. In the remainder of this section, we review the most relevant mobile reflective middleware with respect to the criteria for a failure handling model summarized in Table 2.1.

#### 3.3.4.1 Reflective Middleware for Mobile Computing (ReMMoC)

ReMMoC [GBS03, GBS05] is a web services-based reflective middleware that provides support to dynamically adapt service discovery and communication protocols to deal with the heterogeneity of a mobile environment. It consists of a number of component frameworks, whose structure can be altered by means of reflection. ReMMoC employs the notion of web services to allow mobile clients to be developed independently from the concrete technique used for both the service discovery and communication component frameworks. More specifically, applications employ a generic service lookup API for service discovery and the Web Service Definition Language (WSDL) to invoke operation on remote services. ReMMoC then takes care of mapping such abstract requests to the concrete service discovery and interaction protocols provided in the middleware implementation. As such, applications can discover services in the environment that matches a service type irrespectively of the discovery mechanism that is advertising it, and then interact with a newly discovered service by means of different communication paradigms. By default, ReMMoC incorporates two service discovery protocols (SLP and UPnP lookup), and two communication paradigms (RMI by means of CORBA and SOAP RPC, and publish/subscribe using STEAM).

To the best of our knowledge, ReMMoC lacks abstractions for failure handling. As remarked by the authors in [GBS05], ReMMoC focuses on providing service lookup and communication protocols, and "other features including leasing and service events are not considered". Internally, the ReMMoC implementation does associate operations which can be remotely innovated by other services with a lease [Gra04]. Such a lease is released when the service receives an *endReceive()* operation that stops incoming messages, and flushes the underlying communication component so that operations can be executed on a new binding (which may implement a different communication paradigm). Leases in ReMMoC are, however, not meant to be manipulated by applications, and no information is provided about their nature, e.g., whether or how they can be renewed or revoked.

#### 3.3.4.2 Mobile Platform for Actively Deployable Service (MobiPADS)

MobiPADS [CC03] is a mobile computing middleware built on top of Java that exploits reflection to achieve dynamic adaptation to context changes. The middleware is implemented as a collection of service entities called *mobilets* which are built as services of primitives services forming a service-chain composition. In order to enable dynamic service reconfiguration, the system maintains a system profile that describes the meta-level configuration of the system components and entities. According to the system profile, MobiPADS determines the configuration of the service chain to be executed. Mobile applications can also define a profile to modify the default service reconfiguration mechanism.

Interestingly, MobiPADS supports dynamic adaptation at both the middleware and application layers. At the middleware level, MobiPADS has a built-in event notification mechanism in order to react to contextual events. It directly interacts with the underlying operating system which monitors and subsequently signals the events to the middleware. Mobile applications can also subscribe to a set of contextual events that they are interested in monitoring and react to contextual events. In contrast to other reflective approaches like ReMMoC, the MobiPADS middleware cannot reconfigure a mobile application directly; it can only provide the application with contextual changes. Applications, on the other hand, can directly interact with the underlying middleware by means of its reflective API to request service adaptations (e.g., change the configuration of the service chain or the behaviour of individual mobilets). Contextual events represent information from within the device (including CPU utilization, RAM, storage and power) to resources external to the device (including network status and connectivity status). Such a notification mechanism thus allows reacting to network connectivity (**C3**).

Similarly to Rover, MobiPADS assumes a nomadic network in which clients run on mobile devices, and servers are stationary devices interconnected by a wired network. In this case, communication is not restricted to a client/server model as mobilets can also directly communicate with one another. To the best of our knowledge, MobiPADs relies on WebPADS communication model, which is designed to be asynchronous [CCC02]. However, it does not feature full synchronization decoupling because the service chain reconfiguration mechanism may suspend clients during the synchronization process. In particular, the initiator of the synchronization has to be suspended until the system hosting it and the system hosting other participants is reconfigured.

| | Communication | | | | | | State Consistency | | Memory Management | | Tool Support |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | C1 Decoupled Communication | | | | C2 High-level Representation of Failures | C3 Reacting to Network Connectivity | C4 Local Failure Recovery | C5 Failure Handling Strategies | C6 Relaxing Soundness | C7 Contractual Memory Management | C8 Forcing Failures |
| | Time | Space | Synch. | Arity | | | | | | | |
| **Distributed Programming Languages** | | | | | | | | | | | |
| Argus | No | No | No,blocking futures | No | None, exception raised | No | Only for stable objects | None, built-in transactions | No | No | No |
| Mozart | Yes | No | No, logic vars block | No | Tight to network connectivity | Yes, using fault streams | No | Fix set of distribution parameters | No | No, only if lease strategy | Yes |
| E | No | No | Yes | No | Tight to network connectivity | No, Only to network disconnections | No | None, they could be encoded using handlers at both ends of a reference | No | No | No |
| **Formal Languages for Distributed Computing** | | | | | | | | | | | |
| ABS | Yes | No | Yes | No | Fault names (∼exceptions) | No, Only to (fail-stop) failures | No | Compensating mechanism on client side | No | No | No |
| **Object-Oriented Middleware** | | | | | | | | | | | |
| JINI | No, relies on JavaRMI | Yes | No | No | Lease | No, Only to lease expiration | No | Handler for lease expiration on client side | No | Yes, relies on JavaRMI | Yes |
| Rover | Yes | No | Yes | No | None, failure transparency | Yes, registering a callback | Yes | Either QRPC or RDO | No | No | No |
| DR-OSGi | No | No | No | No | Tight to network connectivity | Yes | Yes | Extensible set of hardening strategies | No | No | No |
| **Tuple Space Middleware** | | | | | | | | | | | |
| LIME | Yes | Yes | Yes | Yes | No | Yes, using reactions | No | None | Yes | No | No |
| TOTA | Yes | Yes | Yes | Yes | No, it could be encoded in propagation rules | Yes | Yes | None, they could be encoded in propagation rules | Yes | No, it could be encoded in propagation rules | No |
| **Publish/Subscribe Middleware** | | | | | | | | | | | |
| EMMA | Only in durable subscriptions | Yes | Yes | Yes | Lease | No | Yes | Either durable or non-durable subscriptions | Yes | Yes | No |
| one.world | Yes | Yes | Yes | Yes | None, built-in recovery | No, only after restoring or migration | No | Either checkpointing or migration | Yes | Yes, using leases | No |
| **Reflective Middleware** | | | | | | | | | | | |
| ReMMoC | Depending on service lookup and communication protocol employed | | | | None | No | No | None | Yes | No | No |
| MobiPADS | Yes | No | No, blocking reconfiguration | Yes | None | Yes | No | None | No | No | No |

Table 3.1: Survey of Related Work.

## 3.4 Discussion of Surveyed Systems

Table 3.1 evaluates the surveyed systems on the criteria for a failure handling model suitable for MANETs summarized in Table 2.1. Throughout this chapter we have discussed the entries in Table 3.1 on a system-by-system basis. Each entry indicates, for a given system under study, whether and how it adheres to the criteria. We now conclude the chapter by evaluating the entries for each criterion individually.

**C1 Decoupled Communication** allows processes to deal with the effects of intermittent disconnections by enabling them (1) to communicate while being disconnected (decoupling in time), (2) to interact anonymously without knowing their exact addresses (decoupling in space), (3) to remain responsive to communication (decoupling in synchronization) and (4) abstract from the concrete number of processes communicating with (arity decoupling).

Time-decoupled communication is supported in many systems by either buffering messages sent while parties are offline (like in Rover), or by communicating by means of an *intermediate coordinator* which also buffers messages until a potential receiver comes online (like some tuple space and publish/subscribe middleware). For example, in TOTA, each tuple space acts as a logically shared data structure that stores tuples injected by (disconnected) processes.

Decoupling in space is a characteristic naturally supported in distributed data-driven models such as tuple space or publish/subscribe systems. However, some object-oriented systems like Jini, employ service discovery mechanism to allow applications to acquire a reference to a remote object without knowing its address.

Decoupling in synchronization is achieved in many systems by either employing an asynchronous message passing style of communication (like E) or by communicating via an intermediate coordinator which is responsible for the actual transmission of messages (like in tuple space and publish/subscribe middleware). Note that tuple space approaches which have not been devised for a mobile setting, usually do not feature decoupling in synchronization because the operations to remove tuples are blocking ( i.e., a process is blocked until a matching tuple becomes available in the tuple space).

Finally, arity decoupling is also inherent to tuple space and publish/subscribe systems since all communication happens in a one-to-many manner. In these systems, the receiver of a tuple or event is potentially *any* process reading from a tuple space, or *any* subscriber that matches the event, respectively.

**C2 High-level Representation of Failures** enables processes to determine when their connections or exchanges of data have terminated. Systems adhering to the classic goal of distribution transparency usually include built-in mechanisms that try to hide network failures from the application level and when it is not possible, failures are represented as exceptions. On the other hand, many systems do provide abstractions for detecting failures (like in Mozart or E) but the representation of failures is coupled with the underlying network connectivity. Finally, other systems, like Jini, limit the relationships between entities in the network by means of leasing. A failure is then represented as the lease expiration. One of the advantages of such a representation of failures is that lease expiration happens independently from the state of the underlying network connectivity.

**C3 Reacting to Network Connectivity** enables processes to monitor the network connectivity of other processes and react if necessary. Reacting to network connectivity is supported in systems targeting the mobile environment as part of the mechanisms designed for context-awareness. For example, MobiPADS provides a notification mechanism that allows applications to react to contextual events, being the connection status of devices in the network one of them. LIME, on the other hand, enables processes to monitor the connection status of underlying systems by means of a read-only system-maintained tuple space. In other systems this is completely aligned with the failure handling mechanisms. For example, Mozart supports reaction to network connectivity of processes by installing fault streams applied on the software entity to monitor.

**C4 Local Failure Recovery** enables processes to recover from failures based on their local state by increasing availability of data. This is achieved in some systems by either providing support for disconnected operations (like in ROVER or DR-OSGi) or by means of an intermediary coordinator which sends messages employing replication-based protocols. For example, in EMMA, durable subscriptions are sent using an asynchronous epidemic routing protocol in which the message is replicated for all reachable hosts. In TOTA, on the other hand, tuples are replicated according to a propagation rule which allows a form of scoping on the tuple dissemination process.

**C5 Application-dependent Failure Handling Strategies** enable processes to define the most appropriate compensating action upon a failure. Systems that provide failure handling features either provide a built-in (set of) strategies to react to failures, or provide handlers to react to failures. Systems that provide built-in failure handling strategies can either apply them by default (e.g., Argus), or allow the application to select the most appropriate strategy among the set of supported strategies (e.g., Mozart, Rover, and EMMA). For example, in Mozart, applications can apply a list of distributed parameters to entities providing a way to determine a strategy for access, state consistency and garbage collection before they get distributed. In systems that provide handlers to react to failures, programmers must manually encode failure handling strategies themselves (e.g., E and TOTA). For example, E handlers triggered when a reference broke could be used to explicitly implement a rebinding strategy to a reference to an object providing an equivalent service that the broken one. A remarkable exception is DR-OSGi which provides a default set of failure handling strategies which can also be extended by deploying OSGi bundles that implements the handler API.

**C6 Relaxing Soundness** enables connections or exchanges of data to remain valid in the presence of intermittent disconnections. This is supported in mainly by data-driven systems such as tuple spaces and publish/subcribe middleware. For example, TOTA relaxes soundness by employing a replication-based model.

**C7 Contractual Memory Management** enables processes to reclaim unused resources in the presence of permanent disconnections by agreeing on the lifetime of the data shared with other processes before it is actually shared. All systems that provide contractual memory management employ leasing, except for TOTA in which this needs to be manually encoded on top of the tuple propagation rules. For example, Jini support contractual memory management due to its reliance on JavaRMI in which leasing have been incorporated to enable the reclamation of objects in the face of failures of client objects. However, many of these systems assume that leases are managed by the underlying system. As such, they do not allow applications to determine the lifetime of the leases (like in one.world), or provide very limited support for manipulating them (like in Java RMI).

**C8 Forcing Failures** enables processes to trigger failure handling even if no physical network failure occurred. This is a central feature to enable the development software tools as it enables them to examine the application behaviour with respect to different network topologies. Yet, it is only supported in Mozart. This supports our observation that the deployment of MANET applications is still in its infancy. Although much mobile computing middleware has been proposed over the last decade (as shown in this survey), very few applications have been developed relying on that middleware. This could also explain why software tools for MANET applications have received very little attention from academy so far.

## 3.5 Conclusion

Based on our analysis of related work, we draw the following major conclusions:

- First, leasing offer a good solution for supporting both a higher-level representation of failures (as only network failures exceeding a certain time interval are considered a failure at the programming level) and contractual memory management (as the lifetime of the data is determined by the lease time interval). Current systems, however, introduce leasing as a low-level concern in their implementation in order to simplify resource management, or to describe the lifetime of exchanged data. As a result, leases are either transparent to the application, or very low-level support is provided to manipulate the default leasing behaviour. This forces developers to manually build abstractions that express different leasing variants and application-dependent failure handling strategies.

- Second, many object-oriented approaches provide mechanisms to express different failure handling strategies based on disconnected operation techniques, but they fail to provide a total decoupling of processes due to its reliance on RPC.

- Finally, tuple space and publish/subscribe middleware specifically designed for mobile computing provide the best decoupling of processes. In particular, publish/subscribe and tuple space-based systems that have been designed for mobile environments provide decoupling in space (as they allow processes to communicate anonymously by means of a tuple space or an event broker) and arity decoupling (since all communication is inherently one-to-many). However, many publish/subscribe systems do not feature full decoupling in time since they do not buffer events sent to offline communicating parties. Tuple space-based middleware built around a federated tuple space model also suffers from this limitation since tuples can only be exchanged when the communication party that issued a tuple space operation is in communication range of the device offering the requested tuple. While most tuple space systems enable a process to monitor the connection status of other remote parties hosting a tuple space by means of a dedicated tuple space or tuples, publish/subscribe systems offer no support for monitoring the connectivity of publishers.

In this work, we propose to devise a failure handling model for MANETs around the concept of leasing in which leases are combined with a decoupled communication model. We aim to provide a high-level representation of leases, rather than consider them as a low-level concern which should be hidden from applications. In order to support application-dependent failure handling strategies, we will pursue a reflective approach such that leases are provided as an extensible framework (as found in reflective middleware and in DR-OSGi) to allow programmers to express different leasing strategies. All this together is what we call *ambient-oriented leasing* which is the topic of Chapter 6 of this dissertation. More concretely, chapter 6 explores the integration of leasing into a distributed object-oriented model that exhibits decoupled communication. In Chapter 7, we explore ambient-oriented leasing in the context of a data-driven model based on tuple spaces. Before we can go into further detail, Chapter 4 introduces AmbientTalk, the language that serves as our language laboratory to investigate ambient-oriented leasing, and Chapter 5 introduces AmbientTalk/M, a dialect of AmbientTalk including a reflective architecture suitable to ease the development of distribution and failure handling abstractions.

# Chapter 4

# Ambient-Oriented Programming in AmbientTalk

The work described in this dissertation builds on the principles of ambient-oriented programming (AmOP) paradigm (described in Section 2.2) and the technical foundation of the AmbientTalk language, a concrete incarnation of the AmOP principles. We employ AmbientTalk as the research vehicle in which ambient-oriented leasing has been designed and implemented. Although our failure handling abstractions could have been implemented in other mobile computing platforms, AmbientTalk already provides a number of built-in primitives that help express them in a more natural way. In particular, it offers a non-blocking communication model that aligns well with the requirements to deal with partial failures. Next to explaining in detail its distribution and communication model, we also describe its reflective capabilities which facilitate the experimentation with novel language features from within the language itself. This chapter gives an introduction to those features of AmbientTalk that are required to understand the code excerpts and technical contributions presented in following chapters. A thorough introduction to the language can be found online [DGM$^+$07].

## 4.1 The AmbientTalk language

The AmbientTalk language described in this chapter is actually AmbientTalk/2, a successor of the original AmbientTalk/1 language presented in Dedecker's dissertation [Ded06]. While both languages have an actor-based event-driven model of concurrency and feature abstractions for service discovery, AmbientTalk/2 features more modular and stratified meta-level programming abstractions. We refer the interested reader to Van Cutsem's dissertation [Van08] for a thorough discussion of the differences between the two versions. For simplicity's sake, in this chapter, we use the term AmbientTalk to refer to AmbientTalk/2, and the language's full names are used when the distinction is necessary.

We first describe the core features of AmbientTalk's kernel including its object-oriented, concurrent and distributed features. Subsequently, we describe AmbientTalk's reflective layer. We illustrate AmbientTalk's features by means of an advertising application that runs on mobile phones. In this application, users can advertise items which are subsequently displayed on the screen of the cell phone of nearby potential customers that have registered their interest in those advertisements. For example, the application can be used to buy or sell concert tickets at the venue itself [EGH06]. This application has been used as running example in previous AmbientTalk publications [VMG$^{+}$07, GSL$^{+}$10].

## 4.2 Object-oriented Programming in AmbientTalk

AmbientTalk is built on object-oriented principles. Its object model is directly based on the prototype-based language Self [US87]: classless slot-based objects can be reused and extended by means of *delegation* [Lie86] (also known as object-based inheritance). Computation is expressed in terms of objects sending messages to other objects or themselves. Listing 4.1 below illustrates object-oriented programming in AmbientTalk in the context of the advertising application. The code defines a prototypical object which represents the advertisements for the items traded in the application. This object is created ex-nihilo[1] and bound to the `Advertisement` variable (line 1). An item object has fields to store the item's state and methods to define useful behaviour, e.g., to get a textual description of the item. The last four lines of code shows how the object can be used to create new advertisements. Sending **`new`** to an object creates a clone of that object which is initialised by invoking the clone's `init` method. The `init` method thus serves as the "constructor" for objects.

Listing 4.1: A prototypical advertisement object.

```
 1  def Advertisement := object: {
 2    def category;
 3    def title;
 4    def description;
 5    def advertiser;
 6    def init(aCategory, aTitle, aDescription, anAdvertiser) {
 7      category := aCategory;
 8      title := aTitle;
 9      description := aDescription
10      advertiser := anAdvertiser;
11    };
12    def getDescription() {
13      "Advertiser: " + advertiser.getContactDetails() + "\n" +
14      "Title: " + title + "\n" + description;
15    };
16  };
17  // instantiate a new ad
18  def anAdvertisement := Advertisement.new(Leisure,
19    "Cheap Tickets",
20    "2 FC Barcelona tickets for tonight 50% off",
21    sender);
22  anAdvertisement.category;
23  anAdvertisement.getDescription();
```

**Message Sending** Messages are used to invoke fields and methods of an object as shown in listing 4.1 in lines 22 and 23, respectively. When an object receives a message, it looks up the method and applies it. Note that the lookup of `category` is treated as a `category()` method[2].

If the receiver object does not understand a message, it *implicitly* delegates the message to its parent object; the object bound to its slot named **`super`**. By default, **`super`** is bound to **`nil`**. If a message is delegated to **`nil`**, then the original receiver

[1] An object created ex-nihilo is also called an anonymous object in AmbientTalk.

[2] In AmbientTalk, the field access is treated as the invocation of a nullary method. This property is also known as the *uniform access principle* [Mey00].

Listing 4.2: Extending the prototypical advertisement object.

```
1  def BookAdvertisement := extend: Advertisement with: {
2    def author;
3    def init(aTitle, anAuthor, aDescription, anAdvertiser) {
4      super^init(Leisure, aTitle, aDescription, anAdvertiser);
5      author := anAuthor;
6    };
7    def ==(aTitle, anAuthor) {
8      (self.title == aTitle).and: { author == anAuthor };
9    };
10 };
```

of the message is informed of the failed lookup. Since **super** is just a regular field of an AmbientTalk object, it can be changed dynamically at runtime enabling *dynamic inheritance* comparable to Self. AmbientTalk provides the **extend:with:** function for specifying that a new object delegates to an existing prototype. Listing 4.2 shows a new prototype `BookAdvertisement` which extends the `Advertisement` prototype. **extend:with:** implies that when the `BookAdvertisement` is cloned, the clone's super field is initialized to a clone of the `Advertisement` parent object including its own copies of the `category`, `title`, `description`, and `advertiser` slots.

AmbientTalk also allows objects to *explicitly* delegate a message to another object by means of the ^ operator[3]. A traditional use of this operator in AmbientTalk is to perform super-sends (akin to Java ones). It is important to remark that when using delegation either implicitly or explicitly, the pseudo-variable **self** is left bound to message sender. For example, the super-send shown in listing 4.2 (line 4) delegates the `init` message to the `Advertisement` object, leaving self bound to the `BookAdvertisement` object.

**Scoping** AmbientTalk is a lexically scoped language, i.e., a name always refers to the environment where it was defined. However, a name can be also looked up in the delegation chain. Hence, an object's scope consist of its lexical scope plus the scope of its parent objects. To distinguish which scope to use when resolving a name, developers need to qualify names to be looked up in the object scope. This means that unqualified names are always resolved in the lexical scope (e.g., `author` in listing 4.2, line 8), while qualified names are always resolved in the receiver's object scope (e.g., `**self**.title` in listing 4.2, line 8).

**Block Closures** AmbientTalk uses block closures to represent *delayed* computations such as implementing the branches of an **if:then:else:** control structure, or nested event handlers (as will be described later). Block closures are constructed using the syntax `{ |args| body }`, where the arguments can be omitted if the block takes no arguments. The following code excerpt shows a typical use of blocks to iterate over an array of advertisements, to show all advertisements on the screen.

```
myAdvertisements.each: { |ad| GUI.show(ad) }
```

Note that AmbientTalk supports both traditional canonical syntax (e.g., `GUI.show(ad)`) as well as keyworded syntax (e.g., `myAdvertisements.each: block`) for message sends and method definitions. As a general rule, keyworded syntax is used for

[3] Read as the carret operator.

control structures (e.g., **`while:do:`**) or object declarations (e.g., **`object:`**), while the canonical syntax is used for expressing application-level behaviour.

**Type Tags** In order to support object classification in a protoype-based and dynamically typed language, AmbientTalk relies on the notion of *type tags*. Type tags are used to categorise objects explicitly by means of a nominal type. They are best compared to empty Java interface types, like the typical "marker" interfaces that are used to tag objects as `java.io.Serializable` and `java.lang.Cloneable`. The following code excerpt illustrates the use of type tags for representing the different categories of advertisements in our running example.

```
deftype Leisure;
deftype Sports <: Leisure;
```

A type tag is declared using the **`deftype`** keyword. It can be a subtype of zero or more other type tags using the `<:` operator. Objects in AmbientTalk may be tagged with zero or more type tags when they are created by means of the **`object:taggedAs`** `:` construct. However, it is not possible to alter the type tags of an object after its creation, i.e., types tags remain constant. This restriction is motivated by the fact that one important use of type tags in AmbientTalk is to provide a description of what kinds of services an object provides to remote objects as we will describe later.

## 4.3 Concurrent Programming in AmbientTalk

AmbientTalk is a concurrent actor-based language [Agh86]. In contrast to AmbientTalk/1, AmbientTalk/2's actors are not modeled as active objects, but rather as *communicating event loops* like in the E programming language. Recall from Section 3.1.3 that in this model, actors are represented as containers of regular objects and computation is expressed in terms of messages enqueued in an actor's mailbox, also called the actor's *message queue*. The actor encapsulates a single thread of execution which perpetually takes messages from its queue and executes the corresponding method on the receiver of the message. The method is then run to completion, which is called a *turn*. A turn is executed atomically, i.e., an actor cannot be suspended or blocked while processing a message. Actors take incoming messages from their message queue one by one (i.e., in serial order) to avoid race conditions on the state of regular objects. Throughout this text, we will use the terms *event loop* and *actor*, and the terms *message queue* and *mailbox* interchangeably in the context of AmbientTalk.

In AmbientTalk, each object is said to be owned by exactly one actor. Only an object's owning actor (also referred to as the *owner actor*) can directly execute one of its methods. In other words, an actor has exclusive access to its mutable state. Objects owned by the same actor communicate using sequential message sending. Objects owned by different actors can only communicate using traditional asynchronous messages to one another by means of *far references*. A far reference is an object reference that spans different actors and only allows asynchronous communication; any attempt to access the object synchronously via a far reference raises a runtime exception. Asynchronous messages sent via far references are delivered to the actor owning the receiver object, more specifically, they are enqueued in the actor's message queue.

Figure 4.1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the actor's event loop thread which perpetually takes messages from its mailbox and processes them. The "stick" arrowhead represents a far reference which is used to communicate asynchronously with an object that is owned by another actor. The figure 4.1 also shows a message send from a customer to a supplier actor in the context of our running example. When a notification object `N` sends a message `getDescription()` to advertisement object `A`, the message is enqueued in the message queue of `A`'s actor which eventually processes it.

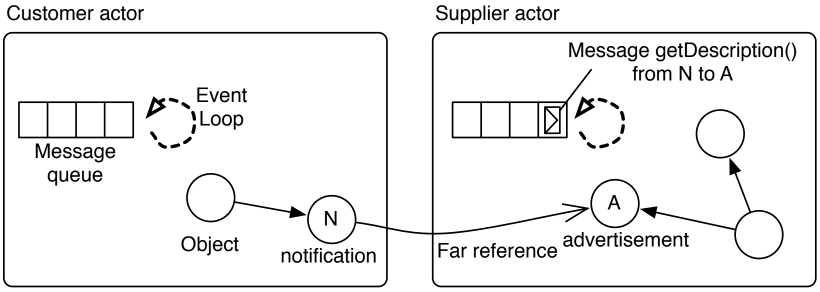


Figure 4.1: AmbientTalk actors as event loops.

### 4.3.1 Message Passing Semantics

AmbientTalk employs different syntax for sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). Asynchronous messages can be sent between objects owned by the same actor or by different actors. When sending an asynchronous message between objects owned by the same actor, the message's arguments are passed *by reference*, exactly as it is the case with standard synchronous message sending. When sending an asynchronous message across different actors, objects are passed *by far reference*: the arguments of the invoked method are replaced by far references to the original objects. In either case, AmbientTalk guarantees that asynchronous messages are delivered to an object in the same order as they were sent. It is important to remark that far references are instead passed by copy to a third actor. This means that the third-party actor can directly communicate with the owner actor of the referenced object. When a far reference is passed back to the object's owning actor, it is resolved into a local reference.

Objects can also be passed by copy in an inter-actor message send if they have been declared *isolates*, i.e., if they have been created with the type tag **`Isolate`**. They are called isolates because they do not have access to their surrounding lexical scope (i.e., they behave as isolated pieces of code). Any lexically visible variable required by an isolate needs to be manually copied into the isolate scope. When an isolate is passed in an asynchronous message, a copy is created and all objects it directly refers are also recursively passed (according to their own semantics). From the receiver actor perspective, this leads to the creation of a clone in that actor whose initial state is the same as the state of the original object at the moment the message was sent. The receiver actor can then operate on the copy synchronously without additional inter-actor communication. In short, isolates differ from regular AmbientTalk objects in two ways: they are passed by copy rather than by far reference in inter-actor message sends, and they cannot use any free lexically visible names.

Listing 4.3: Future-type message passing in AmbientTalk.

```
1 def descriptionFut := advertisement<-getDescription()@FutureMessage;
2 //register an observer on the resolved value of the future
3 when: descriptionFut becomes: { |description|
4   // execution is postponed until future is resolved
5   system.println("New advertisement received: " + description);
6 } catch: { |exception| ... };
7 // code following when: is processed immediately
```

### 4.3.2 Future-type Message Passing

Synchronous and asynchronous message passing also differ in the way return values are obtained. While a synchronous message send is handled as a regular method invocation in which the sender waits for the result of computation, an asynchronous message send returns immediately **`nil`**. In order to avoid the use of explicit callback methods to process the result of an asynchronous computation, AmbientTalk employs the notion of *futures* [YBS86] (also known as *promises*)[4]. A future is a placeholder object for the return value of an asynchronous message send. With the introduction of futures, explicit callbacks are no longer necessary: the future serves as an implicit callback. To illustrate future-type message passing, consider again the advertising application. Customers can use their mobile phone to receive advertisements of nearby devices and can get extra information about the advertisement. Each mobile phone runs an AmbientTalk application consisting of a single actor. Line 1 of listing 4.3 shows how the description of an advertisement can be requested, given that `advertisement` denotes a far reference to the advertisement broadcasted by another actor. The `@FutureMessage` annotation makes the `getDescription` message send immediately return a future (which stored in the variable `descriptionFut`). Once the return value is computed, it “replaces” the future object; the future is then said to be *resolved* with the value. If the asynchronously invoked method raises an exception rather than returning a value, the exception is propagated to the future object; the future is then said to be *ruined* with the exception.

Most of the systems introducing futures support synchronization on the return value (represented by a future) by suspending the thread that tries to access an unresolved future. In AmbientTalk, an actor cannot suspend waiting for a future to be resolved or ruined as this would violate the non-blocking communication principle of an AmOP programming language (cf. Section 2.2). Instead, AmbientTalk employs non-blocking futures as in the E language. Developers can register a block of code with a future, which is executed asynchronously when the future becomes resolved or ruined. For example, the description that supplied the advertisement can only be printed on the screen when the `descriptionFut` future is resolved to a string value as shown in listing 4.3 (lines 3-6). The **`when:becomes:catch:`** function takes a future and two block closures as arguments, and registers the functions as *observers* on the future. If the future is resolved to a proper value, the **`becomes:`** closure is applied to the resolved value (passed as argument). If the future is ruined with an exception, the **`catch:`** closure is applied to the exception. The execution of either of these closures is always scheduled in the owning actor’s mailbox, such that their execution is serialised with respect to other messages processed by the actor. As a result, the code following **`when:becomes`**

[4] Future are actually not built into the AmbientTalk kernel, but implemented by means of the meta-level architecture described in Section 4.6.

**:catch** is always processed immediately even if the future was already resolved or ruined when **when:becomes:catch:** is called.

Note that the return value of the **when:becomes:catch:** function is itself a future. That future is resolved with the return value of either the **becomes:** or the **catch :** closure, or ruined with an exception raised during the execution of either closure. This means that the resolution of the future returned by **when:becomes:catch:** depends on the resolution of another future (i.e., the future passed as argument in **when: becomes:catch:**). This property is known as *future pipelining* (or *future chaining*).

It is important to remark that asynchronous messages can be sent to a future regardless of its state. If the future is unresolved, the future object accumulates the messages it receives. When the future is resolved, accumulated messages are forwarded to the resolved value. Future pipelining will also occur if an asynchronous message sent via a future has itself a future.

## 4.4 Distributed Programming in AmbientTalk

A distributed application consists of multiple parts running on different devices in a network. Each device typically runs a virtual machine that executes that part of the application. In AmbientTalk, each virtual machine is said to *host* one or more actors. Objects are considered to be *remote* when they are owned by different actors, even if those actors are hosted by the same virtual machine. Hence, actors are the unit of concurrency and distribution. The main issues distinguishing distributed programming from concurrent programming are partial failures and service discovery (i.e., a mechanism to acquire a first reference to a remote object). In the remainder of this section we describe the language provisions build atop the event loop concurrency model to deal with these issues.

### 4.4.1 Far References and Network Failures

Since objects residing in different virtual machines are owned by different actors, they can only communicate asynchronously via far references. As such, far references are the only kind of remote object references in AmbientTalk. Far references by default mask partial failures. When a network failure occurs, a far reference becomes *disconnected.* Messages sent to a disconnected reference are buffered until it becomes *reconnected* once the network partition is restored at a later point in time. When the reference becomes reconnected, it sends all accumulated messages to the remote object in the order they were originally sent. This behaviour makes that intermittent failures have no impact on the application's control flow.

In order to allow applications to react to changes on far references's connectivity and to apply failure handling code if necessary, AmbientTalk provides two *failure event handlers* which can be registered on a far reference. They install an observer which is triggered whenever the reference becomes disconnected or reconnected. The code excerpt below shows how the advertising application uses these event handlers to indicate nearby sellers that are currently online.

```
// seller is a far reference to a peer in the advertising application
whenever: seller disconnected: {
  gui.showOffline(seller);
};
whenever: seller reconnected: {
  gui.showOnline(seller);
};
```

The two **whenever:** functions take as argument a far reference and a nullary closure that is applied whenever the interpreter detects the disconnection or reconnection of the referenced object, respectively. Both functions return an object whose `cancel` method can be used to cancel the registration of the observer with the interpreter.

Next to those event handlers, AmbientTalk allows developers to retract the messages accumulated in a far reference by means of the **retract:** construct. It returns a table containing copies of all messages in the queue at the point in time when the function was called and cancels their delivery.

### 4.4.2 Exporting and Discovering Objects

Objects can be *implicitly* exported to other actors when they are passed as arguments or return values in inter-actor message sends (as seen in Section 4.3.1). In order for objects to acquire a first far reference to a remote object, a mechanism is necessary to make objects available in the network. In AmbientTalk, an actor can *explicitly* export objects to other actors by means of a type tag. The code excerpt below shows how an advertisement object can export itself by means of the type tag stored in the advertisement's `category` field.

```
def pub := export: self as: self.category;
```

The **export: as:** function is parameterized with the object to be exported and the type tag by which it is discoverable. It returns an object that can be used to "unexport" the object by invoking `pub.cancel()`. It it important to remark that far references already handed out before calling the `cancel` method remain valid. This implies that an object may still be remotely accessible despite not being discoverable anymore.

From the moment an object is exported in the network, other objects can discover it by registering a discovery event handler with the interpreter. In the advertising application, a user can be notified whenever a leisure advertisement is received as follows:

```
def sub := whenever: Leisure discovered: { |advertisement|
  def descriptionFut := advertisment<-getDescription()@FutureMessage;
  /* see Listing 4.3 */
};
```

The **whenever:discovered:** function takes as arguments a type tag and a unary closure, and registers the closure with the interpreter as a *discovery event handler* for the type tag. Whenever an actor is encountered in the network that exports a matching object, the closure is asynchronously applied taking as argument a reference to the newly discovered object. In this case, `advertisement` is bound to a far reference to an advertisement object owned by another actor. One can then start sending asynchronous messages via the far reference, e.g., the `getDescription()` message to print the description of the advertisement on the screen. An object matches a discovery event handler if its exported type tag is a subtype of the type tag argument of **whenever: discovered:**. Similar to the **export:as:** function, the **whenever:discovered:** function returns an object whose `sub.cancel()` method cancels the registration of the discovery event handler with the interpreter.

## 4.5 Interoperability with Java

AmbientTalk has been built in Java and thus runs on top of the Java Virtual Machine (JVM). It has been designed so that it can also interoperate with Java in a similar

Listing 4.4: Interoperability with Java.

```
1  def swing := jlobby.javax.swing;
2  def frame := swing.JFrame.new("Advertisement");
3  def titleField := swing.JTextField.new(20);
4  def textArea := swing.JTextArea.new();
5  def advertiseButton := swing.JButton.new("Advertise!");
6
7  advertiseButton.addActionListener( object: {
8    def actionPerformed(actionEvent) {
9        def title := titlefield.getText();
10       def content := textArea.getText();
11       def advertisment := Advertisement.new(theCategory, title, content, self);
12       export: advertisement as: advertisement.category;
13   };
14 });
```

way to other dynamic languages implemented on top of the JVM such as Groovy, and JRuby. This means that AmbientTalk objects can access all Java libraries available in the underlying JVM. For the purpose of this dissertation, it is important to know that Java objects and classes can be instantiated or invoked from within AmbientTalk using regular AmbientTalk syntax, and that there are built-in conversions between primitive data types of Java and AmbientTalk. We briefly illustrate those concepts by means of the graphical user interface (GUI) for the advertisement application. Further details on the mechanism that allows the interoperability of AmbientTalk with Java can be found elsewhere [CMM09, GSL$^+$10].

Listing 4.4 shows a small part of the GUI for the advertisement application using the Java Swing framework. The GUI consists of a simple input field for the title of the advertisement, a text area used for the description of the advertisement and a button to publish the advertisement. The `jlobby` variable (line 1) gives access to the Java packages available in the underlying JVM. A Java object (e.g., `advertiseButton`) is represented in AmbientTalk as a regular object whose fields and methods correspond to the public interface-level fields and methods in the Java object. Java classes are also represented as regular objects whose fields and methods correspond to the public static fields and methods in the Java class. When AmbientTalk code invokes a Java method that expects an argument typed as an interface, any AmbientTalk object can be passed to that method. For example, in line 7, an AmbientTalk object is passed as argument in the call to `addActionListener`. Such an anonymous object plays the role of a Java `ActionListener` object and as such, it implements the `actionPerformed` method. Whenever the user presses the `advertiseButton`, the anonymous object will be notified by the underlying system.

This concludes the explanation about the AmbientTalk kernel. We now turn our attention to its reflective facilities.

## 4.6 Reflective Programming in AmbientTalk

Reflection allows programs to observe and modify their own structure and behaviour at runtime [Smi84]. Through the use of reflection, the kernel language can be extended with both programming support and new language constructs. This is usually accomplished by means of *reification*: the “materialization” of the interpreter structure in the language. AmbientTalk reifies both its object and actor model with the goal to serve as language laboratory for facilitating such experiments in the context of MANETs. The reflective architecture of AmbientTalk is based on *mirrors* [BU04], meta-level objects that allow one to reason about (base-level) objects and make reflective code independent of a particular implementation. Mirrors are *causally connected* [Mae87] to the objects they mirror. If the object is changed by base-level code, the changes can be observed via the mirror. Conversely, changes applied to the object via the mirror, modify the actual object.

Mirrors traditionally provide support for *introspection* (the ability to inspect the structure and behaviour of a program), *invocation* (the ability of a program to dynamically execute program fragments), and *self-modification* (the ability of a program to change its own structure). In AmbientTalk, such mirrors are named explicit mirrors. In addition, AmbientTalk provides implicit mirrors which combine mirror-based reflection with *intercession* (the ability to change the program behaviour). An implicit mirror is a mirror used by the interpreter itself when performing meta-level operations on base-level objects. In the remainder of this section, we describe both explicit and implicit mirrors on objects and actors.

### 4.6.1 Mirror-based Reflection using Explicit Mirrors

In AmbientTalk, explicit mirrors are modeled after the mirrors present in Self [ABC$^+$00] and Strongtalk [BG93]. Mirrors on objects are created by means of the **`reflect:`** construct. It consults a mirror factory to create a mirror on the given object. This ensures that a mirror is not accessed directly from the base-level object it reflects on. Listing 4.5 (line 1) shows how a mirror on the `Advertisement` object introduced in listing 4.1 is acquired. A mirror represents the object on which it reflects as a collection of slots. A slot binds a name to a method. An object cannot contain two or more slots with the same name. AmbientTalk represents fields by a pair of accessor and mutator slots. The accessor is a method that returns the value of the field. The mutator is a method that takes as argument the value to assign to the field.

Listing 4.5 also shows the three forms of reflection that explicit mirrors support. First, introspection allows the retrieval of object's slots (line 2). Second, invocation allows the explicit invocation of object's slots (lines 3-5). An identifier prefixed with a backquote (') denotes a symbol. The arguments passed to `invoke` denote a receiver and an *invocation*, an object encapsulating the selector (a symbol) and the actual arguments (a table). Finally, self-modification allows the addition or removal of slots (lines 6-8). The `createMethod` helper function creates a method given a name, parameters, body of the method and annotations (metadata of a method).

```
1 def mirrorOnAd := (reflect: Advertisement);
2 mirrorOnAd.listSlots().map: { |slot| slot.name }; // `[category,title,init,...]
3 mirrorOnAd.grabSlot(`category); // accessor for field category
4 mirrorOnAd.grabSlot(`category:=); // mutator for field category
5 mirrorOnAd.invoke(Advertisement, createInvocation(`getDescription, []));
6 def method := createMethod(`getAdvertiser, [], `{ advertiser;}, []);
7 mirrorOnAd.addSlot(method);
8 mirrorOnAd.removeSlot(`getAdvertiser);
```

Listing 4.5: Introspection, invocation and self-modification via explicit object mirrors.

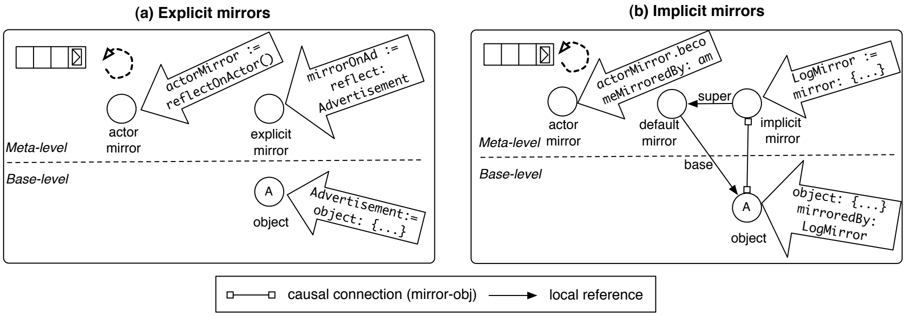

(a) Explicit mirrors
actorMirror := reflectOnActor()
mirrorOnAd := reflect: Advertisement
actor mirror
explicit mirror
Meta-level
Base-level
A
object
Advertisement:= object: {...}
(b) Implicit mirrors
actorMirror.becomeMirroredBy: am
LogMirror := mirror: {...}
super
actor mirror
default mirror
implicit mirror
Meta-level
Base-level
base
A
object
object: {...} mirroredBy: LogMirror
causal connection (mirror-obj)
local reference

Figure 4.2: AmbientTalk actors from a reflective perspective; representation of (a) explicit mirrors, and (b) implicit mirrors.

**Explicit Mirrors on Actors** As previously explained, each object is owned by one actor. To be more precise, each actor owns both the base-level objects (representing an application) and their meta-level objects (mirroring base-level objects). In addition, each actor also owns an *actor mirror*, a special mirror denoting the mirror on the actor as a whole. This mirror differs from the explicit mirrors just described in that it does not reflect on a single base-level object, but rather on the entire event loop. All objects owned by the actor share the actor mirror. Figure 4.2 (a) gives an overview of the different objects owned by an actor given the definition of the `Advertisement` object and its `mirrorOnAd` mirror (shown in listing 4.1 and 4.5, respectively). The top-level function **`reflectOnActor`** returns the explicit mirror on the actor executing the call.

The actor mirror reifies those operations which transcend the scope of a single object or are related to inter-actor operations, e.g., communication among remote objects. As shown in listing 4.6, the explicit actor mirror also supports three forms of reflection supported by explicit mirrors on objects: introspection, invocation, and self-modification. Those forms of reflection take a different shape when applied to actors. First, introspection (line 2-3) allows developers to inspect the actor's mailbox, and the service discovery publications and subscriptions. `listIncomingLetters` returns a copy of the actor's mailbox represented as an array of letter objects. A letter consists of a receiver-message pair together with a method named `cancel` which can be invoked to remove the letter from the actual actor's mailbox. Second, invocation (line 4) has been adapted for asynchronous message passing and as such, it enables the explicit sending or reception of asynchronous messages. The `createMessage` method returns an asynchronous message with the given name, parameters and annotations. Finally, self-modification (line 5) allows developers to alter the service discovery mechanism, i.e., add new publications or subscriptions.

Listing 4.6: Introspection, invocation and self-modification via explicit actor mirrors.

```
1  def actorMirror := reflectOnActor();
2  def mailbox := actorMirror.listIncommingLetters();
3  mailbox.map: {|letter| letter.message.selector};
4  actorMirror.send(adverstisement, actor.createMessage(‘getDescription, [], []));
5  actorMirror.publish(anAdvertisement, anAdvertisement.category);
```

Listing 4.7: A prototypical logging mirror.

```
1 def LogMirror := extend: defaultMirror with: {
2     def invoke(delegate, invocation) {
3       system.println("invoked " + invocation.selector + " on " + self.base);
4       super^invoke(delegate, invocation);
5     };
6   }
7 }
```

### 4.6.2 Mirror-based Intercession using Implicit Mirrors

AmbientTalk's mirror-based architecture goes beyond introspection and also allows objects to modify the default semantics of the language. An AmbientTalk object can be associated to an *implicit* mirror which describes the object's semantics. That is to say that the implicit mirror provides custom semantics for the default meta-level operations on a base-level object. The implicit mirror is then used by the interpreter when manipulating the base-level object.

Listing 4.7 shows a mirror for logging all methods invoked on an object. The mirror encodes the logging behaviour by overriding the default implementation of the `invoke` meta-level operation. The top-level variable `defaultMirror` refers to an implicit mirror object containing AmbientTalk's default metaobject protocol. Although any object can serve as an implicit mirror as long as it provides a complete implementation of the metaobject protocol, most implicit mirrors extend the default mirror to implement their custom semantics. In this case, the `LogMirror` does not override any meta-level operation except for `invoke`.

As also shown in Listing 4.7 (line 3), a mirror can refer to the base-level object it mirrors by means of its `base` slot. Note that the `LogMirror` mirror has not yet been causally connected to any base-level object i.e., it has not been absorbed by the interpreter. A mirror becomes causally connected to an object when the object itself declares to be mirrored by the mirror. This happens when an object is created by means of the **`object:mirroredBy:`** construct. In AmbientTalk, a base-level object causally connected to an implicit mirror is called a *mirage* object. Listing 4.8 redefines the `Advertisement` prototype from listing 4.1 as a mirage whose (meta) behaviour is now described by the `LogMirror`. The **`object:mirroredBy:`** construct is parameterized with two closures: a closure to create the mirage, and one for the mirror construction. The interpreter creates a mirage in three steps. First, an empty mirage object is created. Second, an implicit mirror is created and associated with the empty mirage which is passed as argument to the mirror construction closure (`newMirage`). Finally, the empty mirage is associated with its implicit mirror and its initialisation code is executed. From this point on, the mirage and its mirror are causally connected and the mirror is effectively used by the interpreter. Note that there is a strict one-to-one

Listing 4.8: Definition of a mirage.

```
def Advertisement := object:{
  /* the original implementation */
} mirroredBy: { |newMirage| LogMirror.new(newMirage)};
```

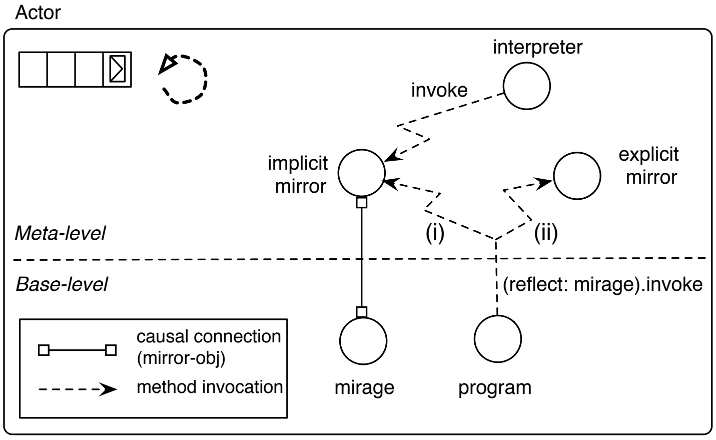


Figure 4.3: Relationship between explicit and implicit mirrors on mirages. When a program reflects on a mirage, it consults the mirror factory which returns the implicit mirror (by default), or (ii) another explicit mirror (if a custom factory is used).

relationship between a mirage and its unique implicit mirror, i.e., the causal connection remains constant.

Figure 4.2(b) shows the resulting `Advertisement` mirage causally connected to its implicit mirror (called `LogMirror` in the figure). The top-level function **`mirror:`** is syntactic sugar for creating an extension of the default mirror ( i.e., it is equivalent to line 1 of listing 4.7). Once a mirage is created, it is indistinguishable from a regular object. Developers can reflect on them by means of the **`reflect:`** construct previously explained. Figure 4.3 illustrates the relationship between mirages, implicit and explicit mirrors. When the interpreter manipulates the mirage, it always does it via its implicit mirror. However, when programs want to introspect the mirage via the **`reflect:`** construct, they first consult the mirror factory. The default mirror factory returns the implicit mirror, but it can also return another explicit mirror if a custom mirror factory was installed. More details on the mirror factory can be found elsewhere [MVT$^+$09].

**Implicit Mirrors on Actors** Developers can also install implicit mirrors on actors by means of the **`becomeMirroredBy:`** meta actor operation (as also shown in Figure 4.2(b)). In contrast to implicit mirrors on objects, implicit mirrors on actors can be installed *at any time* during the lifetime of the actor by any object owned by the actor.

Listing 4.9 defines and installs an implicit actor mirror which provides a custom implementation for the mirror factory on objects. The actor mirror defines a method named `createMirror` which serves as a factory method for the creation of explicit mirrors on objects. Listing 4.9 defines and installs a customized mirror factory in order to create “sealed objects”. To this end, the `sealedMirrorFactory` actor mirror overrides the default implementation of `createMirror` to return an explicit mirror which seals the base-level object (lines 4-10). The returned mirror overrides the default implementation of the `addSlot` and `removeSlot` meta-level operations such that adding or removing slots to an object raises an exception (lines 6-9). Note that the `sealedMirrorFactory` actor mirror extends the current explicit actor mirror such that it inherits the implementation for all other meta-level operations.

Listing 4.9: An implicit actor mirror.

```
def actorMirror := reflectOnActor();
def sealedMirrorFactory := extend: actorMirror with:{
  //override createMirror to provide a custom mirror factory for objects
  def createMirror(onObject) {
    def defaultExplicitMirror := super^createMirror(onObject);
    extend: defaultExplicitMirror with:{
      def addSlot(slot){raise: IllegalOperation.new("Cannot add slot")};
      def removeSlot(slotName){raise: IllegalOperation.new("Cannot remove slot")};
    }
  }
};
actorMirror.becomeMirroredBy: sealedMirrorFactory;
```

Having discussed both explicit and implicit reflection on both object and actors, we give an overview of the AmbientTalk's metaobject protocol in the next section.

### 4.6.3 AmbientTalk's Metaobject Protocol

The metaobject protocol (MOP) of AmbientTalk can be divided into a set of independent protocols reifying different aspects of objects and actors to meta programmers. Table 4.1 provides a brief overview and description of the various protocols. We categorize the different protocols according to the entity which implements the protocol API. A protocol API is exposed by object mirrors, actor mirrors, or it consists of a number of methods defined in object and actor mirrors. The full semantics and signatures of the meta-methods of the protocols can be found elsewhere [MVT$^+$09, DGM$^+$07]. In the remainder of this section, we detail the most important protocols for the purposes of this dissertation: the message invocation protocol and the object marshalling protocol.

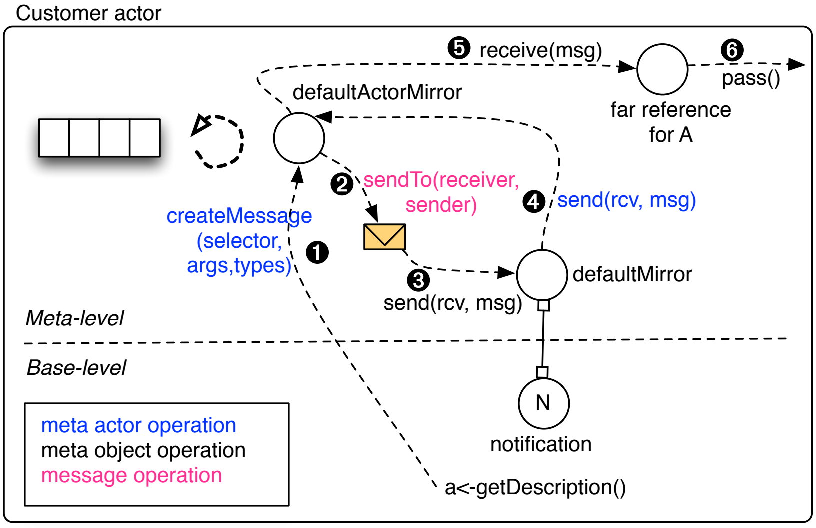


Figure 4.4: Message invocation and object marshalling protocol at sender side.

We explain the message invocation and object marshalling protocols by means of the `getDescription()` asynchronous message sent from a customer to a supplier actor shown in Figure 4.1. Figures 4.4 and 4.5 provide a graphical illustration of the sequence of meta operations as a result of executing the `advertisement←getDescription()` expression. We first explain the sequence of calls of the message invocation protocol at the sender side by means of Figure 4.4. A call to a meta-method corresponds to a step of the protocol (which is annotated with the corresponding number in the figure).

**Step 1.** As a result of evaluating the `advertisement←getDescription()` expression, the `createMessage` method is invoked on the actor mirror. The method returns a message object which consists of a selector, arguments and type tags. `create Message` can be overridden to add additional metadata to a message object, e.g., the `FutureMessage` annotation used for future-type message passing.

**Step 2**. Upon creation a message does not have a receiver yet. When the message send expression is evaluated, the receiver is set and the `sendTo` method on the message is called. The default implementation of `sendTo` calls the `send` meta operation on the receiver's object mirror.

**Step 3.** The `send` meta method on an object mirror delegates the responsibility of sending the message to the actor mirror.

**Step 4.** The `send` meta method on the actor mirror schedules the transmission of the message by making the far reference for the receiver object `receive` the message. If the receiver object lives in the same actor (i.e., it is a local object), the actor directly calls the object's mirror `receive` method.

**Step 5.** When a far reference receives a message, it enqueues the message in its internal message queue. The interpreter then triggers the object marshalling protocol by calling `pass` on the message, and all the objects reachable from its arguments.

**Step 6.** The semantics of `pass` is akin to the `writeReplace` method in Java: it allows an object to determine how it should be passed to remote objects. For regular AmbientTalk objects, the default implementation of `pass` asks the actor mirror to create and return a far reference designating the referenced object by invoking `createReference` meta operation. For objects declared as isolates, the implementation of `pass` returns a copy of that object. This concludes the sequence of meta-operations at the sender side.

After the serialization, the message awaits in the far reference message queue its turn to reach the beginning of the queue. When this happens, the serialized message is dequeued and physically transmitted to the receiver actor. Figure 4.5 illustrates the sequence of call of the message invocation and object marshalling protocols at the receiver side.

**Step 1.** When a message is received from the network, the interpreter first triggers the object marshalling protocol. The unmarshalling of objects is reified by means of the `resolve` method whose semantics is akin to `readResolve` method in Java: it returns an object replacing the unmarshalled object. The default implementation of `resolve` just returns the object previously marshalled.

**Step 2.** Once `resolve` has been called to the message (and all objects reachable from its arguments), the interpreter triggers the message invocation protocol at the receiver side by asking the actor mirror to `receive` the unmarshalled message.

**Step 3.** The actor mirror in turn asks the receiver object mirror to `receive` the message. The receive method in the actor mirror allows developers to alter the seman-

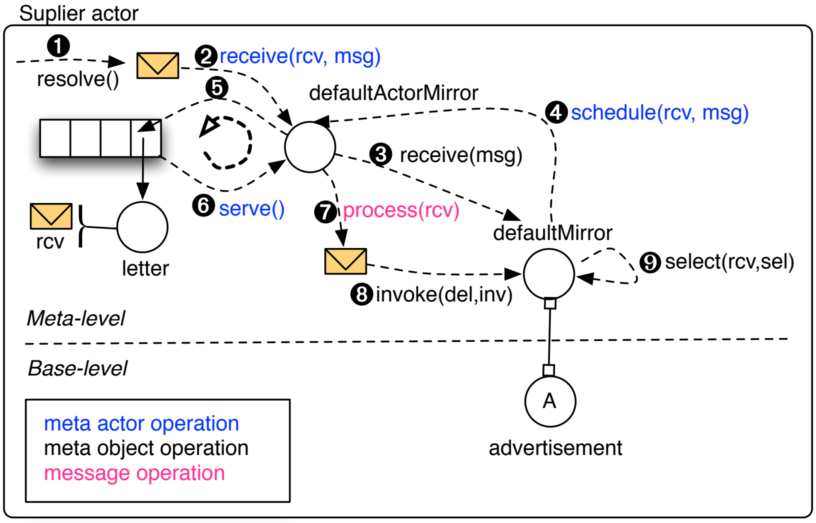


Figure 4.5: Message invocation and object marshalling protocol at receiver side.

tics of the reception of *all* messages sent from other actors to objects owned by an actor.

**Step 4.** By default, the object's mirror `receive` method delegates the reception of the message back to the actor mirror by invoking `schedule`.

**Step 5.** The `schedule` method adds the incoming receiver-message pair as a letter in the actor's mailbox.

**Step 6.** When the letter reaches the beginning of the queue, it will be dequeued by means of the `serve` method. The default implementation of `serve` calls `process` on the message object of the dequeued letter. By overriding `serve`, a metaprogrammer can redefine the message processing behaviour of an actor, e.g., to be prioritised.

**Step 7.** The `process` method of a message makes the receiver object's mirror invoke the method corresponding to the selector. By default, an asynchronously received message is thus transformed into a regular (synchronous) method invocation (in this example `advertisement.getDescription()`).

**Step 8.** The `invoke` method first obtains the closure of the slot corresponding to the selector by means of the `select` method, and applies it. If a matching slot for the selector is not found in the object nor in one of the objects of its delegation chain, the `doesNotUnderstand` meta method is invoked returning an exception.

We conclude our explanation of AmbientTalk's message invocation protocol with a note on messages. As the reader may have noticed, messages in AmbientTalk are first-class objects. AmbientTalk distinguishes between three kinds of messages: asynchronous ones (expressed as `←m()`), synchronous ones (expressed as `.m()`) and delegated ones (expressed as `^m()`). The `<+` operator sends a first-class message to a receiver object as if the message was invoked on the receiver in the source code. The operator expresses either an asynchronous, a synchronous or a delegated message based on the type of the given message. For example, the `advertisement←getDescription()` expression is equivalent to the following code snippet:

```
def mesage := <-getDescription();
advertisement <+ mesage;
```

## 4.7 Conclusion

In this chapter, we have described AmbientTalk, a distributed object-oriented programming language that was especially designed to ease the development of applications that run on mobile devices interacting over a wireless network. To this end, the language has been designed along the principles of the ambient-oriented programming. The language's asynchronous concurrency model mask failures by default, allowing applications to remain responsive upon network disconnections. The languages built-in publish/subscribe engine allows objects to discover one another in a peer-to-peer manner, without depending on any centralised infrastructure.

We have also extensively described AmbientTalk's support for reflection. Reflection is an integral part of the language given its role as a language laboratory to experiment with novel AmOP language features. In the following chapter, we discuss limitations of AmbientTalk's meta-level architecture, which motivate the need for a revised architecture. Subsequently, we introduce AmbientTalk/M, a descendent of AmbientTalk which extends its meta-level architecture with tools to ease the development of failure handling abstractions and tool support reflectively.

Table 4.1: Overview of the metaobject protocol of AmbientTalk.

| | |
|---|---|
| *Protocols exposed by object mirrors* | |
| Structural Access Protocol | Reifies the structure of an object as a collection of slots. It consists of methods to add and remove slots, get access to a first-class slot representation, and lists all *owned* slots ( i.e., slots bound directly in the object, and not in one of the objects in the delegation chain). |
| Object Instantiation Protocol | Reifies the act of creating new objects from existing objects. It consists of two methods: `clone` which creates a (shallow) copy of the object, and `newInstance` which first calls the `clone` method to create a clone of the current object and then initializes the clone by invoking its `init` method. |
| Type Tag Protocol | Reifies the type tags attached to an object. It consists of two methods: `isTaggedAs` which tests whether an object is (transitively) tagged with a type, and `listTypeTags` which returns an array of type tags which the object is tagged. |
| Relational Testing Protocol | Reifies the relationship between objects. It consists of three methods, called `isCloneOf`, and `isExtensionOf` and `isRelatedTo`, which test whether two objects are related through cloning, object extension, or a combination of cloning and extension, respectively. |
| Evaluation Protocol | Reifies the evaluation and quotation of abstract grammar objects. It consists of two methods, called `eval` and `quote` which allow for evaluating and quoting an object part of a parse tree, respectively. Regular objects simply return themselves upon quotation and evaluation, but explicit parse-tree objects (e.g., method invocation) have dedicated evaluation functions. |
| *Protocols exposed by actor mirrors* | |
| Service Discovery Protocol | Reifies the act of publishing and discovering remote objects. It consists of methods to advertise a local object as a service, register a discovery event handler, and to list the advertised local objects and the active discovery event handlers. |
| Mirror Creation Protocol | Reifies the act of creating mirrors. It consists of two methods which serve as the mirror factory for explicit mirrors on objects and actors: `createMirror` which returns an explicit mirror on an objects, and `getExplicitActorMirror` which returns an explicit on the actor. In addition, there is a method to install a new actor mirror in the actor. |
| *Protocols exposed jointly by actor and object mirrors* | |
| Message Invocation Protocol | Reifies both asynchronous and synchronous method invocations. It consists of a number of methods in the object mirror to intercept the invocation of synchronous messages, the sending of asynchronous messages, and the reception of both asynchronous and synchronous messages. In addition, it provides methods in the actor mirror to create and send asynchronous messages, and to control the actor mailbox (the queue of incoming receiver-message pairs buffered before being processed). |
| Object Marshalling Protocol | Reifies the act of marshalling and unmarshalling objects when they are passed as arguments in inter-actor messages. It consists of two methods exposed by object mirrors: `pass` which allows an object to determine how it should be passed to other objects, and `resolve` which determines how it should be resolved upon arrival. In addition, there is a method in the actor mirror, called `createReference`, which returns a far reference to a local object. |

# Chapter 5

# Enhancing Meta-level Engineering in AmbientTalk

In the previous chapter, we have described AmbientTalk as an object-oriented, distributed programming language with a meta-level architecture based on the principles of mirror-based reflection. Using such a meta-level architecture has a number of benefits with regard to the implementation of new language features. First, the implementation of a language feature is *encapsulated* in a dedicated meta-level entity, increasing reusability. Second, the meta-level entity implementing a language feature is *stratified* with respect to the base-level code ensuring that it does not interfere with application-level code. Finally, since base and meta-level code are separated in different layers it also makes it easier to enable reflection only when really required.

This chapter discusses AmbientTalk/M, a dialect of AmbientTalk that improves its reflective architecture with tools to ease the development of distribution and failure handling abstractions. AmbientTalk/M revisits mirror-based reflection in the context of an ambient-oriented language based on event loops, and contributes two concepts: a new representation of remote object references as a pair of metaobjects representing both ends of a reference, and the introduction of an observer mechanism into mirrors that allows metaprogrammers to react dynamically at runtime to the manipulation of an object by the interpreter.

## 5.1 Motivation

The motivation for a revisited meta-level engineering is based on a number of shortcomings when using a mirror-based architecture for building (1) communication-oriented abstractions for MANETs and (2) tool support in the form of a distributed debugger. What makes reflection[1] attractive in the design of distributed mobile systems is that it improves flexibility [Caz01]: it helps developers deal with the openness and dynamic nature of MANETs (e.g., the same application may behave differently depending on the underlying network topology, location, user preferences, etc.). A reflective approach exposes implementation details of the distributed system and allows developers to manipulate them and provide their own solutions to a number of issues such as com-

[1] In this case the term reflection refers to *behavioural* reflection, the ability of a program to access and modify a dynamic representation of itself at runtime.

munication, synchronization, object serialization, failure handling, etc. Furthermore, reflection provides means to achieve a clear separation of concerns [McA95, TNCC03]. As such, it facilitates experiments with different distributed abstractions while avoiding interfering with the application code. Thanks to those strengths, a lot of research effort has been devoted to implement distributed and concurrent problems using reflection and meta programming techniques [McA95, Caz01, GGM94].

At first sight, it may seem that AmbientTalk's reflective architecture is sufficiently powerful to deal with the development of distributed abstractions since it reifies both the actor and object model. However, in this section, we illustrate shortcomings arising from the implementation of distributed abstractions and tool support for which a mirror-based reflective architecture does not provide simple and clear solutions. We distinguish between shortcomings on (1) the reification of language constructs for distribution ( i.e., structure correspondence), (2) the representation of distributed communication, and (3) implicit mirrors.

### 5.1.1 Limitations of Structural Correspondence in Ambient-oriented Programming

In a mirror-based architecture, the principle of structural correspondence [BU04] states that each element of the base-level language maps to an interface in the reflective API. When applying a mirror-based architecture to a distributed object-oriented language, the reflective layer should reify both sequential and distributed parts of the language. As such, AmbientTalk's reflective layer supports reflection on objects and actors (the unit of distribution). However, as we show in this section, some distributed constructs of the language lack a meta-level representation. This is partially due to the fact that there is little prior work that exposes as language constructs distributed concerns such as service discovery. Moreover, most work in reflection on actors has been conducted in languages or frameworks that model actors as ABCL/1-like active objects, rather than as mere containers for regular objects as is the case in the communicating event loop model of E and AmbientTalk. We further distinguish shortcomings in the reification of the communication traces, object marshalling and the environmental context.

**Communication Traces.** Many classic actor frameworks and languages provide the notion of a *meta-actor* [DA07]. Typically meta-actors only allow developers to intercept incoming and outgoing messages to alter the behaviour of the actor system. AmbientTalk/1 introduced the concept of mailboxes that capture the history of messages, i.e., the outgoing messages that have been acknowledged to be received by another actor, and the messages that have been processed. AmbientTalk/1's novelty with respect to other active object models is that these mailboxes are reified as first-class passive objects that can be manipulated by the programmer. Those mailboxes make explicit the communication traces of the actor and allow programmers to properly recover from an inconsistent state due to disconnections, e.g., by reversing (part of) the application's computation.

Nevertheless, AmbientTalk/2's reflective architecture does not provide explicit representation of the communication history of an actor. This is a consequence of the fact that in AmbientTalk/2's concurrency model replaces the notion of actors as active objects with the notion of actors as event loops. In an event loop concurrency model, actors become containers of passive objects which can be remotely referenced by other actors by means of far references. As a result, the communication traces of an actor are now represented by the actor message queue together with the set of far references that an actor holds. We identify three main issues in such a representation of communication traces:

- The message queue of an actor represents the messages that have been received by the actor but not processed yet, i.e., the actor's incoming messages. While the message queue is reified in the actor's reflective layer, the message history is not represented. Obtaining messages already processed boils down to overriding the `invoke` meta operation on an object mirror. However, one cannot obtain a message history dynamically as an implicit mirror on an object can only be installed when the object is created. We will further elaborate on the repercussion of that limitation later in this section.

- Rather than having an outgoing message queue, outgoing messages are contained within far references. The main reason for this is that passive objects owned by other actors can be individually designated by means of far references. Thus, each far reference contains a part of the outgoing communication of an actor. Yet, no support is provided at the reflective layer to manage the whole set of far references that an actor owns. This is necessary to build distributed abstractions such as network-aware references [PHGD11, PHD11]. Although one can override the **`whenever:discovered:`** operation to intercept the reception of a far reference to a newly discovered object, it is not possible to intercept far references acquired implicitly by passing objects as arguments in asynchronous messages. As a result of this limitation the implementation of network-aware references in AmbientTalk is not fully reflective, but required modifications on the interpreter.

- Finally, obtaining messages that have been acknowledged to be received boils down to creating a custom referencing abstraction that exchanges extra messages between the two ends of the reference. Although one can dynamically override the `createReference` meta operation to install the new referencing strategy, it will not be possible to alter all references handed out previously. As a result, meta programs observing the communication traces of an actor (e.g., debuggers, monitors, etc.) cannot be dynamically coupled to a running application.

**Object Marshalling.** AmbientTalk's reflective layer includes an object marshalling protocol that reifies the act of exporting an object to another actor. It allows developers to replace an object by a different one when it crosses the actor boundaries. Although similar protocols can be found in mainstream languages like Java, AmbientTalk exposes object marshalling as a meta-level concern separated from base-level code by means of object mirrors. The object marshalling protocol, however, is only partially reified at the actor level for pass-by-reference objects[2]. While the act of marshalling objects is explicitly reified, developers cannot trap the act of un-marshalling objects at the receiver side, forcing developers to create custom referencing abstractions to encode this.

It is also important to point out the implications of an event loop model in the representation of object marshalling. In an object-oriented distributed model such as in JavaRMI, object marshalling happens when a client object executes a remote method invocation (since communication is synchronous). In an event loop actor-based model

[2] Unless specified otherwise, we refer to pass-by-reference objects as objects.

such as in AmbientTalk, object marshalling happens when an asynchronous message is added to the actor's outgoing mailbox; in the case of AmbientTalk, when it is added to the far reference's message queue. However, the object may actually be transmitted long after it is marshalled. For example, the message may be accumulated in the far reference's queue due to a disconnection. While these semantics are simple and applicable for a wide range of objects, some objects (such as reactive values or time changing values like leases) require to reflect over the possible changes of its fields between the marshalling point and the transmission point. Unfortunately, a representation of object marshalling at the transmission point is not available in AmbientTalk's reflective layer.

**Environmental Context.** In an ambient-oriented language reifying the environmental context of distributed interactions allows developers to react to changes of the underlying network connectivity. AmbientTalk exposes the environmental context by means of the **`when:disconnected:`** and **`when:reconnected:`** primitives (cf. Section 4.4.1). In order to build language features that require monitoring changes in the communication process, e.g., to alter the default semantics of network disconnections, developers need to install those handlers on individual far references. This is, however, a too fine-grained interface to build language constructs that require altering the environmental context of the whole actor. For example, unit testing or simulation frameworks typically need to disconnect an actor as a whole to make the system trigger failure handling code. To circumvent the lack of representation of the environmental context at the actor meta level, developers need to write bookkeeping code to address all references at once.

Although the actor meta level lacks a representation of the established references to an object, AmbientTalk provides a low-level primitive called `takeOffline:` that invalidates all far references to an object, allowing developers to influence the reachability of a remote object. The operation removes an object from the internal data structures of the interpreter that keep track of which objects are remotely accessible, enabling the object to be garbage collected once it is no longer locally referenced. However, there is no counterpart operation reifying the addition of a local object to the set of remote objects (which results in the creation of a far reference for a local object). This is useful for metaprograms observing the distributed behaviour of an actor as it enables them to control the far references being handed out to other actors. Developers can, however, reconstruct the references that are actually handed out by manipulating the meta actor protocol. Recall from Section 4.6.3 that `createReference` is called on the actor mirror *each time* an object is passed to another actor. In its default implementation, the interpreter only creates a far reference the first time it is called and stores it in its internal data structures. In order to monitor which references are actually handed out, programmers need to override this operation and write bookkeeping code that mimics how the interpreter maintains the internal data structures for far references.

### 5.1.2 Limitations of the Representation of Distributed Communication

In order to build distributed abstractions in an object-oriented paradigm, developers typically manipulate distributed communication by implementing custom semantics for remote object references. Most traditional distributed systems expose a remote object reference by a *proxy*[3] in the client object's space which represents a remote object. Such a proxy forwards messages to the remote object it represents. It also has an associated message handler that provides an API to intercept the method invocation performed on it, e.g., the `doesNotUnderstand` method in OpenTalk [ope] (Visualworks SmallTalk's distributed framework), or `invoke` method in `java.lang.reflect.Proxy`.

In AmbientTalk, proxy objects are implemented using mirror-based reflection. A custom object reference is represented by an empty mirage object whose message reception semantics has been modified by an implicit mirror. The implicit mirror wraps a native far reference that is used for the actual message transmission to the remote object. This approach has proven to be useful for developing distributed abstractions that focus on the sender perspective such as futures (described in Section 4.3.1). A future is nothing but a custom object reference that intercepts asynchronous messages sent to the return value. In addition, such a representation of object references preserves stratification since both the sender and receiver object of a reference are un-affected by the meta-level code. As a concrete example, consider again the advertising application from Chapter 4. Recall from listing 4.3 how the `getDescription` message is used to request the description of an advertisement broadcast by a remote object, and print it on the screen. When executing `advertisement<-getDescription()`, the `getDescription` message will not be misinterpreted as part of the meta-level operations of the future.

Although proxies are a simple and widely adopted technique for implementing distributed abstractions, they suffer from a number of inherent limitations. The major problem is that the proxy and the object it represents are distinct entities when conceptually there is only one, giving rise to the *two-body problem* [Eug06]. While this distinction is useful as it preserves stratification, it hampers the implementation of distributed abstractions that take into account the receiver perspective in several ways. First, the base-level code may circumvent a language construct's interface (implemented in the proxy) and expose the internal representation of the original object. This is also known as the encapsulation problem [Eug06]. When building referencing abstractions it is particularly harmful because an object reference defines the kind of access right a client object has on a remote object. As a result, it is not possible to ensure that client objects *always* use the proxy (the custom object reference) to access the remote object. This can pose a number of problems, e.g., a reference to a remote object may be handed out to unwanted or untrustworthy third parties. Second, when a method of the actual remote object returns a self-reference, there is no way for the proxy to intercept that, and again return a proxy. This is known as the "self problem", first described by Lieberman in [Lie86]). This is problematic for e.g., access control abstractions as it should be possible to intercept all invocations. Welch and Stroud discuss other disadvantages of using proxies in [WS99]. An important issue they point out is the implication of inheritance in proxies. In order for the proxy and the original object to be interchangeable, the type hierarchy of the proxy should reflect the type hierarchy of the base-level object it represents. In short, the separation of proxy and remote object introduces problems for method call interception, encapsulation and typing.

Research in reflection has proposed two ways to circumvent those issues: either by merging the object and its proxy as done in Kava [WS00], or by making the proxy inherit from the target object, as in the MOP used by ProActive [CKV98]. In the former approach, the issue is then to dynamically enable proxies on a per-object or even

[3] A proxy is often referred to as an interception object or wrapper.

per-use basis. In the latter, one loses the ability of using chaining to compose different distributed abstractions in favor of using inheritance. The advantage of using chaining is that a meta interception mechanism is separated from the meta-level structuring mechanism [WS99].

Nevertheless, the root cause of the problems with a proxy approach is that modeling distributed communications constitutes a more complex task than intercepting messages in the client object's space. Many distributed abstractions require a representation that reifies both sides of a distributed communication, and allows performing intercession and introspection of the exchanged messages. For example, some abstractions require applying actions both before a message is sent (e.g., to encrypt, compress, attach access policy, etc.) and before the receiver executes it (e.g., to decrypt, decompress, check access lists, etc.). Using proxies to build such abstractions forces programmers to implement two proxies (one at each side) and manually to coordinate their work.

### 5.1.3 Limitations of Mirages

Implicit mirrors are the fundamental building blocks in a mirror-based architecture to implement new language features. Yet, implicit mirrors in AmbientTalk are installed at object creation time and cannot be changed during the lifetime of objects. This limitation is motivated by performance reasons as it enables the interpreter to in-line the meta-level methods associated with objects without an implicit mirror. More importantly, it enforces the one-to-one relationship between a mirage object and its implicit mirror [MVT$^+$09]. However, it is too restrictive for certain types of meta-programs that require activating some functionality at runtime. In particular, software tools like debuggers or object inspectors typically monitor or inspect arbitrary objects *on-demand*. For example, in Smalltalk, programmers can inspect objects of a running application.

To overcome this limitation, one can resort to proxies but we have already pointed out their limitations in the previous section. Alternatively, one can integrate the new language feature using the implicit mirrors on actors instead. Recall from Section 4.6.2 that implicit mirrors on actors can be dynamically installed at any time during the lifetime of actors. However, this is often too coarse-grained and requires developers to derive the operations that need to be trapped manually. As a concrete example consider an object inspector; *only* the methods executed on the object being inspected are relevant to be trapped. Using an actor mirror introduces the overhead of trapping the base-level operations for *all* objects in the actor. In addition, the actor's mirror does not reflect all operations executed on implicit mirrors on objects. For example, since synchronous method invocation is not trapped by the actor mirror, a debugger would not be able to update its GUI whenever an assignment changes the value of an object's field. To circumvent this, metaprogramers are forced to periodically query an explicit mirror on an object for information about the object state.

### 5.1.4 Summary

In this section we have discussed several shortcomings of the mirror-based reflective architecture of AmbientTalk for building distributed abstractions and tools to support the development of ambient-oriented applications which we now summarize.

**Structural Correspondence.** While a mirror-based architecture provides a modular and stratified design, we show that AmbientTalk's reflective layer lacks explicit representation of distributed elements necessary for expressing useful distributed abstractions and tools such as network-aware references. This is due to the fact that when employing mirror-based reflection in the meta-level facilities of a distributed object-oriented event-driven language, true structural correspondence requires an explicit representation of communication traces of an actor, and a representation of object marshalling and the environmental context both at the actor and object level.

**Representation of Distributed Communication.** Representing remote object references by proxies introduces a number of well-defined problems, in particular, for encapsulation and method call interception. In addition, the proxy approach is too restrictive to model distributed communication since only the client perspective is reified, forcing developers to write from scratch abstractions that take into account the receiver perspective.

**Mirages.** Although it is useful for tools like debuggers to react upon the manipulation of an object, this can only be achieved in AmbientTalk by means of implicit mirrors. However, this makes it impossible for those meta-programs to enable some functionalities *on-demand*.

In the remainder of this chapter, we describe AmbientTalk/M, a dialect of AmbientTalk which extends its mirror-based architecture with the following support that solves those issues.

- First, AmbientTalk/M revisits the object and actor reflective layer to provide true structural correspondence for distribution.

- Second, it introduces a new representation of far references based on the *transmitter-receptor model* that reifies distributed communication by opening up the concept of remote object reference. Remote object references are exposed as first-class values with a custom metaobject protocol reified by two metaobjects, one at each end of the reference. In order to avoid re-introducing the inherent limitations of representing object references as mirages, receptors are causally connected to the base-level object they represent for the dynamic context of a distributed interaction. This enables receptors to intercept all method invocations (including self-invocations) as a result of an asynchronous message delivered by a receptor, while preserving encapsulation.

- Finally, it introduces an observer mechanism into actor mirrors which allows developers to get notified of the manipulation of an object by the interpreter without requiring an implicit mirror. As a result, developers can be notified of the execution of a meta operation without having to override it. In addition, an observer can be registered to an (meta or base-level) object dynamically at any time during the lifetime of the actor.

Note that AmbientTalk/M provides a concrete instantiation of these meta-level mechanisms, but their contribution transcends AmbientTalk. In the case of the transmitter-receptor model, to the best of our knowledge, no distributed reflective framework has previously applied mirror-based reflection to the concept of remote object references and dealt with the dynamic context of mobile interactions.

## 5.2 First-Class References as Transmitter-Receptor Pairs

This section describes the design of remote object references using the transmitter-receptor model. The model combines ideas from classic communication-oriented reflective frameworks in which both ends of the communication are reified, with the principles of mirror-based reflection. The main idea is that an object reference is reified into two metaobjects encapsulating all aspects of interactions between senders and receivers. Those metaobjects have been specially designed to reify the distribution aspects of communication in a mobile network, and provide the necessary meta-level hooks to build families of referencing abstractions independently from the base-level functionality of the objects they connect. By manipulating these two metaobjects, developers can handle remote interactions between two objects in a *modular way* as they encapsulate the whole distributed behaviour of a remote object and the references that are handed to client objects.

### 5.2.1 First-class References Overview

A remote object reference denotes a unidirectional communication link that carries messages from a client object to a remotely accessible object (known as a *service* object). Recall from Section 4.4.1 that in AmbientTalk the only type of remote object references are *far references*: references that only allow asynchronous messages to the service object and mask network failures by default. The terms *remote object reference* and *far reference* are thus interchangeable in the context of AmbientTalk. Figure 5.1 shows the conceptual representation of a far reference with a dotted line.

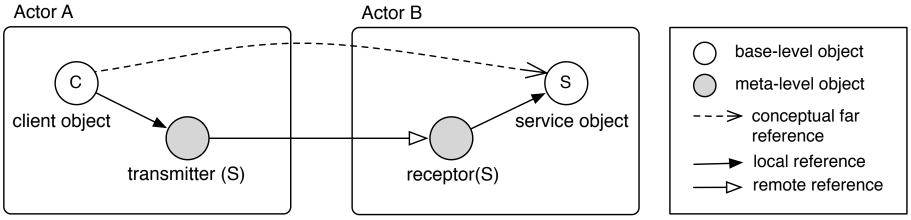


Figure 5.1: Remote object reference model

AmbientTalk/M represents remote object references as first-class objects which expose the communication context of a remote interaction at both ends. In Mark Miller's dissertation [Mil06] we find a simple yet insightful observation for this reification: "a reference is an arrow, and an arrow has two ends". The *source* of a reference denotes the end located in the client object's actor, and the *target* of a reference denotes the end located at the service object's actor. As also shown in Figure 5.1, a far reference is thus reified as a pair of metaobjects, named *transmitter* and *receptor*, representing the source and the target of a far reference, respectively.

**The Receptor** is an object that wraps a service object and whose main purpose is to control distributed interactions with the service object including its message sending, message reception and parameter passing semantics. Each service object is bound to at least one receptor. Not only does the receptor intercept each asynchronous message received by its associated service object in a transparent way, but it also intercepts messages sent as a consequence of the message being processed. The receptor can then perform actions before or after the service object sends or receives a message to handle aspects such as persistence, replication, security, etc. In addition, the receptor also controls the parameter passing semantics of the service object, and can then perform actions according to its semantics, e.g., to ensure that the far reference semantics are not circumvented, limit the lifetime of the reference, etc.

**The Transmitter** is an object that reifies the communication channel at the client side and whose main purpose is to control how the client object sends messages to the service object. Each transmitter is bound to at least one receptor by default. A transmitter is transparently created on the client device when a receptor is unmarshalled, and is used to transmit asynchronous messages to the service object (via the receptor). A transmitter can perform some actions before or after a message send to handle communication aspects such as providing one-to-many communication, applying delivery guarantees, logging successful message sends, etc. In addition, a transmitter exposes the network connectivity of the physical communication with the device hosting the service object, i.e., it reifies the environmental context of the interaction between communicating parties. According to its semantics, it then applies an appropriate failure handling strategy, e.g., buffering messages during disconnections, rebinding to an equivalent receptor, etc. Finally, a transmitter controls the parameter passing semantics of the service object to third party objects.

In general, a remote object reference consists of a transmitter-receptor pair[4] which is responsible for all the distributed concerns between two sides of the reference. Metaprogrammers can build custom object references by modifying the metaobject protocol of transmitters and receptors. Before describing their reflective API, we discuss our reference model in the light of the principles of mirror-based reflection.

#### 5.2.1.1 Mirror-based reflection and the Transmitter-Receptor Model

To adhere to the principles of mirror-based reflection, transmitters and receptors should be cleanly separated from regular base-level code. A receptor is typically hidden from base-level code since it is transparently created and managed by the interpreter, and only metaprograms can access a receptor by invoking a meta-level operation on the base-level object (as we describe later). However, transmitters are used by client objects when sending messages via a reference as shown in Figure 5.1. In order to support a stratified design, it is important to avoid client objects to directly access transmitters. To this end, transmitters are causally connected with an empty object representing the reference at the base level.

Figure 5.2 illustrates our remote object reference model from a reflective perspective. Client objects do not directly hold a local reference to the transmitter (the meta-level representation of a reference), but rather to the base-level object associated to the transmitter. We term such a base-level object the *reference data type*. The reference data type is just a regular empty AmbientTalk object whose behaviour is completely defined by its associated transmitter, i.e., the interpreter uses the transmitter to manipulate it. Developers can obtain access to the transmitter by means of a dedicated

[4] Conceptually, there is a one-to-one correspondence between transmitters and receptors for the default far references. At implementation level, due to performance reasons, there exist at most one receptor and transmitter per actor for a given service object.

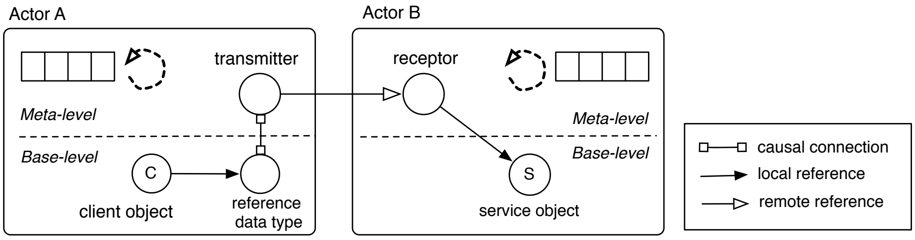


Figure 5.2: Remote object reference model from a reflective perspective

operation as shown below.

```
def transmitter := reflectOnReference: reference;
```

The **`reflectOnReference:`** construct returns the transmitter for the given reference data type. Although a transmitter could be seen as an explicit mirror on a reference data type, **`reflectOnReference:`** does not consult a *transmitter factory*, in contrast to the **`reflect:`** construct for objects. Transmitters and receptors are meant to be used to *specialise* the default far references with custom semantics, i.e., support implicit reflection on far references. As such, we do not make the distinction between explicit and implicit transmitters (nor receptors).

### 5.2.2 First-class References Reflective API

Transmitters and receptors are represented as regular AmbientTalk objects that inherit from the transmitter and receptor prototype encapsulating the default language semantics for far references. Table 5.1, Table 5.2 and Table 5.3 provides a comprehensive overview of the API exposed by the transmitter and receptor prototypes. We categorize the API according to three different protocols which reify the outgoing communication traces, the parameter passing semantics of the target object, and the environmental context of a remote interaction, respectively. These methods may be invoked either by metaprograms that implement custom referencing abstractions, or by the interpreter itself. In the remainder of this section we briefly discuss each protocol.

#### 5.2.2.1 Asynchronous message process protocol

The asynchronous message process protocol reifies the act of processing an asynchronous message via a far reference which consists of two separate phases: 1) receiving and transmitting an asynchronous message from a client object, and 2) receiving and processing the asynchronous message by the target object. These phases are exposed by the transmitter and receptor API, respectively, as shown in Table 5.1. Conceptually, it forms part of the message invocation protocol explained in the previous chapter (Section 4.6.3). In particular, it reifies the final stage of message invocation protocol at the sender side (after step 5 in Figure 4.4), and the alters the way how a message is delivered and processed to the service object at the receiver side (steps in Figure 4.5).

**Transmitter API.** Recall from Section 4.6.3 that when a client object `c` executes `s←m()`, the actor's mirror hosting `c` delegates the responsibility of sending the message to the far reference to the service object `s`. In AmbientTalk-M, the asynchronous

Table 5.1: Asynchronous message process protocol API.

| | |
|---|---|
| *Transmitter API* | |
| `schedule(rcv,msg)` | Adds a letter containing the message and the receiver to the mailbox. |
| `serve()` | Dequeues a letter from the mailbox and transmits it. |
| `transmit(msg)` | Invoked after a message is marshalled, before it is physically transmitted. Returns the message to be transmitted to the receptor of a transmitter. |
| `leave(msg)` | Invoked when a message has been successfully transmitted to the receiver. |
| `listOutgoingLetters()` | Returns an array of all letters currently in the mailbox. |
| *Receptor API* | |
| `accept(msg)` | Makes the service object accept the delivery of an asynchronous message. |
| `sendMessage(rcv,msg)` | Invoked when the service object sends an asynchronous message to a receiver object. |
| `performInvocation(rcv,inv)` | Invoked when the service object performs a synchronous method invocation on a receiver object. |

Listing 5.1: Accessing and modifying a far reference's mailbox

```
def retract: reference matching: closure {
  def mailbox := (reflectOnReference: reference).listOutgoingLetters();
  mailbox := mailbox.filter: { | letter | closure(letter.message)};
  mailbox.each: {|letter| letter.cancel()};
  mailbox;
};
```

message is received by the far reference data type which delegates this task to its transmitter which is asked to `schedule` the message in its mailbox. The transmitter 's mailbox stores asynchronous messages before being transmitted. It works similarly to the actor's mailbox, but it stores outgoing letters rather than incoming letters. An outgoing letter consists of a receiver-message pair together with the marshalled representation of the message. The default implementation of `schedule` first marshalls the message, and then creates a new outgoing letter and buffers it in its mailbox. By overriding this method, a metaprogrammer can redefine message sending scheduling and add additional metadata to a message (e.g., use encryption techniques).

The `listOutgoingLetters` method returns a copy of the mailbox as an array of outgoing letter objects. Similar to `listIncomingLetters` described in Section 4.6.1, this array is not causally connected to the real mailbox of the transmitter. In order to cancel the transmission of a message, one needs to invoke the letter's `cancel` method. Note that the interpreter guarantees that no messages are added to or removed from the transmitter 's mailbox while a meta-program is executing one of these methods. However, the `cancel` method may not succeed if the letter was already put into the physical communication channel. As a concrete usage example, consider listing 5.1 that shows the implementation of a variation of the **retract:** construct (cf. Section 4.4.1) using transmitters. The **retract:matching:** function takes a reference data type and a closure as its arguments, and returns an array of letters of which transmission has been cancelled. It first gets a copy of the mailbox of a transmitter, and filters those messages for which the given closure returns true (lines 1-2). The function then cancels the transmission of the matching letters and returns the array of removed letters (lines 3-4). This construct can be used to avoid the propagation of outdated data during a disconnection, e.g., to remove the messages that update a remote reactive value during a disconnection, so that only the most recent value is sent upon reconnection.

Outgoing letters are transmitted in a serial manner so that successive asynchronous messages sent by a client object to the same service object are delivered in a FIFO order. The serve method is invoked by the interpreter whenever a transmitter should transmit a message from its mailbox. The default implementation works analogously to the actor mirror's serve method: it dequeues a letter object from the mailbox and transmits its message to the receptor associated with the transmitter. By overriding serve, a metaprogrammer can redefine the message transmission to e.g., prioritize message transmission, cancel the transmission of certain messages, etc. Before the message is transmitted, the transmit method is called. This method allows developers to reflect possible changes on the message between the marshalling point (when schedule is called) and the transmission itself, e.g., cancel messages which are outdated, update the message delivery guarantees, etc. Once the message is actually transmitted and has been acknowledged to be received, the leave method is invoked. In this case, overriding leave allows metaprogrammers to perform some actions on the communication history of a reference, e.g., maintaining a log of sent messages. This method concludes the sending phase of the asynchronous message process protocol.

**Receptor API.** At a later point in time, when the actor hosting the service object (s in our example) receives the asynchronous message, the corresponding receptor is asked to accept the asynchronous message to s. The act of accepting a message does not imply that the message will be processed, but rather that the message has arrived at the target. To accept a message thus involves determining what to do with it. The default implementation makes the associated service object receive the message. It can be overridden to perform control actions on the message before the target object process it, for example, to apply decryption on a message, check the permissions of the sender object, implement persistence, etc. It is also possible that the message is not delivered to the target object. For example, if the message is tagged as ControlMessage, it is immediately processed by the receptor.

Recall from Section 4.6.3 that an asynchronously received message is transformed into a synchronous method invocation, i.e., s.m(). In AmbientTalk/M, during the method invocation of an asynchronous message, any other asynchronous message send or a synchronous method invocation is first delegated to the receptor that delivered the executing method invocation. This means that the service object mirror calls sendMessage or performInvocation in the receptor whenever it needs to send an asynchronous or perform a synchronous method invocation, respectively. The default implementation for those operations corresponds to AmbientTalk's default semantics: sendMessage calls the send meta-operation on the service's object mirror, and performInvocation calls invoke method on the receiver object's mirror. Those operations allow developers to control the full extent of the execution of asynchronous messages that receptors deliver to service objects, and prevent them to be executed if necessary. A metaprogrammer can override these operations to, e.g., apply access control properties to the computation resulting from the processing of an asynchronous message, log causal relationships between asynchronous message sends, etc. Listing 5.2 shows how the performInvocation method on a receptor can be overridden to build a *safe reference*, a far reference which does not allow asynchronous messages that cause side effects on the service object.

Listing 5.2: Example of a custom receptor

```
def makeSafeReference(){
  extend: defaultReceptor with: {
    def performInvocation(rcv, inv){
      if: !(invocation.selector.isAssignmentSymbol()) then:{
        super^performInvocation(rcv,inv);
      }
    }
  }
}
```

The custom receptor overrides the performInvocation method only to allow method invocations that do not change an object's state. In AmbientTalk an assignment is expressed as a synchronous method invocation whose selector denotes the assignment symbol for a received field name, e.g., `fieldName:=. The isAssignmentSymbol helper function checks whether the recipient selector of the method invocation corresponds to an assignment symbol.

In many cases, a metaprogrammer wants to share control information between transmitters and receptors transparently to the client and service objects. In order to facilitate this, we introduced the type tag ControlMessage which can be used to annotate messages which are understood by receptors and transmitters and are not further propagated to the target base-level object. If an incoming asynchronous message is annotated with @ControlMessage, it is processed by the meta-level, either a transmitter or receptor object depending on which end of the references receives it. If an incoming asynchronous message is not annotated with @ControlMessage, its processing follows the language semantics described above, i.e., it is handled by either the schedule or accept method of the corresponding transmitter or receptor depending on which end of the conceptual reference receives it. Intercepting the reception of control messages on transmitters and receptors can be done by means of the usual AmbientTalk reflective operations, i.e., installing an implicit mirror on the custom transmitter or receptor.

#### 5.2.2.2 Reference marshalling protocol

The reference marshalling protocol reifies the act of marshalling and unmarshalling objects when they are passed as argument of a message sent to another actor, or passed via the service discovery mechanism. This is reified by means of the marshallingStrategy and unmarshallingStrategy methods on both the transmitter and receptor objects representing a far reference as shown in Table 5.2. In what follows we describe their default semantics, and their relevance for building different referencing abstractions.

Recall from the previous chapter (Section 4.6.3) that marshalling is also reified at the level of the object that is parameter-passed by means of the pass and resolve meta methods. For objects who did not declare themselves to be pass-by-copy, the default implementation of pass asks the actor to create and return a receptor to the object. The receptor is then passed in place of the service object. When this happens,

Table 5.2: Reference marshalling protocol API.

| *Transmitter and Receptor API* | |
|---|---|
| `marshallingStrategy()` | Invoked when a receptor or a transmitter is marshalled. Returns either the receptor or the transmitter to be marshalled instead of this receptor or transmitter . |
| `unmarshallingStrategy()` | Invoked when a receptor or a transmitter is unmarshalled. Returns the object replacing the unmarshalled transmitter or receptor . |

the `marshallingStrategy` method is invoked on the receptor which by default returns the receptor itself. This implies that all client objects share the same receptor to access the service object in the default implementation. The `marshallingStrategy` method at the receptor allows metaprogrammers to redefine how to access the service object by passing a different receptor object. Metaprogrammers can enforce different values of arity of the transmitter-receptor relation:

**unary** at most one receptor is marshalled per service object at any point in time.

**multiary** a separate receptor is marshalled each time the service object is parameter passed.

**n-ary** there is an upper bound on the number of receptors for the service object.

In addition, this method can be used to introduce abstractions to regulate the access to the receptors as the sponsorship mechanism provided by .NET Remoting framework for leases on remote objects [Low03].

When a receptor is received in the client object virtual machine, the interpreter invokes the `unmarshallingStrategy` method. The default implementation asks the actor to create and return a transmitter. The transmitter is bound to the receptor, and returned to the client object. Metaprogrammers can override the operation to redefine the transmitter semantics for a given receptor. This is useful in referencing abstractions in which the transmitter and the receptor need to share control information, e.g., encrypted referencing abstractions.

A client object `c` can also pass an acquired reference (to a service object `s`) to a third-party object `p` in an asynchronous message send, e.g., `p←m(s)`. In that case, the `marshallingStrategy` and `unmarshallingStrategy` methods are invoked by a transmitter object that `c` holds for the service object `s`. Figure 5.3(a) illustrates the default semantics of parameter passing a transmitter. The third-party object `p` acquires a copy of the transmitter object which the original client object held (`c` in the figure). The new transmitter is bound to the same receptor as the transmitter being passed. Note that if the third party actor is the actor hosting the service object, a local reference to the service object is returned instead (since the service object is no longer accessed remotely).

By overriding these two methods, a metaprogrammer can redefine how third parties get access to the service object. In some cases, it may be useful to mediate the access to the service object via the actor who received directly the far reference, rather

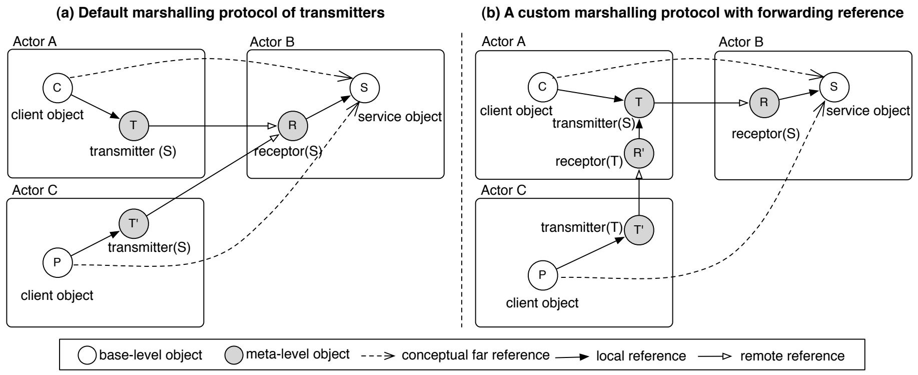


Figure 5.3: Default and custom marshalling protocol of transmitters

than access directly the receptor of the service object. Examples of this pattern can be found in communication abstractions such as network-aware references [PHGD11, PHD11], and passive replication mechanisms as the ones described in [GGM94]. This can be achieved by overriding the transmitter `marshallingStrategy` method to leave behind a *forwarding* reference to itself as shown in Figure 5.3(b). In short, the `marshallingStrategy` method creates and returns a new receptor `R'`, which upon unmarshalling returns a new transmitter `T'`.

To sum up, the `marshallingStrategy` and `unmarshallingStrategy` methods on a receptor reify how a client object acquires a far reference to a service object from *the owner actor*, i.e., the actor hosting the service object. In contrast, the `marshallingStrategy` and `unmarshallingStrategy` methods on a transmitter reify how a client object acquires a far reference to a service object from *an acquaintance of the owner actor*. We will show later in Section 5.2.4 how these methods have been used to implement a concrete referencing abstraction.

#### 5.2.2.3 Environmental context protocol

The environmental context protocol reifies the changes of the underlying network connection with the service object. As shown in Table 5.3, it consists of one method called `addObserver` that can be invoked to register an observer that is notified when the reference state changes to the given state. The native states are identified by the type tags `connected` and **`disconnected`**. By default, when a reference is created, it is set to the connected state. Instead of registering an observer object, a first-class asynchronous message is registered together with the observer object. When a far reference changes to the given state, the first-class message is sent to the given receiver object. As in AmbientTalk/1, we call these first-class messages *observer messages*. As Dedecker already pointed out [Ded06], subscribing messages instead of objects provides flexibility since the interface of the observer object does not need to be fixed with regard to the notification protocol.

Observer messages are scheduled in the recipient actor as regular asynchronous messages. While this simplifies its processing, the observer object may not be able to respond to the change in a timely fashion. Applications subject to time constraints could override the actor scheduling protocol to ensure that these messages are processed before all other messages. Observer messages are tagged by default with the

Table 5.3: Environmental context protocol API.

| *Transmitter and Receptor API* | |
|---|---|
| addObserver(rcv,msg,typeTag) | Registers a message to be sent to the receiver object when the reference is in the state denoted by the given type tag. Returns a subscription object whose cancel method can be used to cancel future notifications. |

@ObserverMessage tag allowing metaprogrammers to distinguish them from functional asynchronous messages sent by the (meta) program itself. Messages used by the interpreter to trigger delayed computations for other sorts of events (e.g., service discovery subscriptions) have also been annotated with an @ObserverMessage tag.

### 5.2.3 Deploying Referencing Abstractions

Any object can serve as transmitter or receptor as long as it provides a complete implementation of the far references meta protocol previously described. The transmitter and receptor prototype objects described in the previous section are accessible by means of the defaultReceptor and defaultTransmitter prototypical objects. These objects are defined in the actor's top-level scope and their purpose is to facilitate the construction of custom referencing abstractions that require only slight variations with respect to default semantics. A custom receptor r can be installed by invoking **becomeReferencedBy:** on the service object's mirror.

Listing 5.3 illustrates the installation of a simple receptor that overrides the accept method to log incoming messages. The **becomeReferencedBy:** method has been added to the explicit mirrors on objects to reify the referencing strategy by which the recipient object will be remotely accessible. When the obj is going to be passed to another actor, the interpreter will do it according to the semantics defined by the given receptor. In this code snippet, it is used to install a new receptor for the obj object that logs all asynchronous messages it processes from client objects. The target variable is defined in the defaultReceptor prototype and refers to the (base-level) service object a receptor wraps.

Listing 5.3: Installing a custom receptor

```
(reflect: obj).becomeReferencedBy: ( extend: defaultReceptor with: {
  //override accept to log the message
  def accept(msg) {
    log("accepting" + msg.selector + " on " + self.target );
    super^accept(msg);
  }
});
```

The **becomeReferencedBy:** meta operation can be applied to any object except for transmitters and receptors . We opted to disallow it to enforce that they are parameter-passed according to their reference marshalling protocol (described in Sec-

Listing 5.4: Prototype implementation of an omireference using a custom transmitter

```
def omnireference: typeTag {
  extend: defaultTransmitter with: {
    def receivers := Set.new(); //to store available matching objects
    def schedule(msg){
      receivers.each: { |rcv|
        rcv <+ msg;
      };
    };
    whenever: typeTag discovered: { |potentialReceiver|
      receivers.add(potentialReceiver);
    };
  };
};
```

tion 5.2.2.2)[5]. In general, the transmitter-receptor pair needs to be *semantically coherent* in order to implement a certain reference strategy. In other words, the protocols implemented by transmitter and receptor need to be composable. Custom transmitters are installed by overriding the `marshallingStrategy` method of the receptor to return a new transmitter . The transmitter to be used is thus determined by the code of the receptor. Such a design enables the modularization of all distributed behaviour of the service object and the referencing abstraction itself into transmitter-receptor pairs.

Note that the fact that the transmitter-receptor pair needs to be semantically coherent does not imply that metaprogrammers need to implement both a custom transmitter and receptor object to build a referencing abstraction. Our framework has been specially designed to ease the development of custom referencing abstractions that require changes on both sides of a reference. However, some families of referencing abstractions only require changes on either transmitter and receptor without requiring a matching counterpart, i.e., the changes provide orthogonal functionalities. For example, this is exhibited by referencing abstractions exploiting one-to-many communication patterns in which the target objects are actually a group of service objects including M2MI handles [KB02], and ambient references [Van08]. The code snippet below shows how to create and use a reference abstraction that transparently discovers and binds to all objects of a certain type (the `Player` type tag). This behaviour is reminiscent of an M2MI omnihandle.

```
def nearbyPlayers := omnireference: Players;
nearbyPlayers<-updatePosition(myPlayerInfo);
```

In this example, an omnireference is used in the context of a mobile game to broadcast a player position to nearby players. Whenever a remote object matching the `Players` type tag is discovered, it is added to the set of objects that the omniference binds to, called the *receivers* of the omnireference. As usual, communication with an omnireference happens by means of asynchronous message passing as shown in the second line of the code snippet. Implementing such an abstraction does not require a custom receptor to regulate the access to the receiver objects; it is sufficient to install a custom transmitter that keeps track of the receivers at the client side of the reference.

Listing 5.4 shows a prototype implementation of such a referencing abstraction. An omnireference extends the default transmitter so as to apply two changes. First, it registers a service discovery request to listen for remote objects exported with the given

[5] As in AmbientTalk, a disallowed operation provokes a runtime exception.

service tag (lines 9-11). Second, when a client object sends an asynchronous message through the reference, the `schedule` method is overridden to forward the message to each receiver (lines 4-8).

### 5.2.4 Case Study: Lazy References

In this section, we illustrate how to use our framework for custom object references by means of a concrete referencing abstraction, namely *lazy references*. Lazy references introduce *lazy parameter passing semantics* [LJE08, Eug03] into AmbientTalk's asynchronous distributed object model. In most state-of-the-art distributed systems objects representing large entities are passed by reference (to avoid wasting network resources with their transfer) and "small" objects by copy (to avoid overhead of remote invocations). Lazy parameter passing provides support for a family of objects which are between these two extremes that require developers to make a trade-off between the overhead of transferring them and their use in a method invocation. This technique allows an object to be first passed by reference, and then transferred by copy at a later point in time if it is required in the execution of a method.

To illustrate lazy parameter passing more concretely, consider an adaptation of the scenario of a music sharing application in [Eug03] that runs on mobile devices. When two people using the music player enter one another's personal area network, the music players exchange their music library's list (not yet the songs themselves). Listing 5.5 below shows the prototype definition of a song object in AmbientTalk. A song object consists of metadata associated with a song (the title, and the artist), and the actual audio track. It has three methods: `init` (i.e., the object constructor), == that ensures equality of songs based on their metadata, and `play` that we explain later.

When a song is exchanged with other music players, a music player receives a copy of the song together with a copy of its meta data so that the end-user can check the title and the artist. In contrast, the audio track is parameter-passed by *lazy reference*. Only when the song is about to be played by the music player, is the audio track copy actually transferred. To this end, the audio track is created by means of `lazyIsolate:` function as shown in listing 5.6. An audio track consists of the array of bytes encoding the audio data, and an `android.media.AudioTrack` instance used to represent an audio resource in an Android mobile device. Its `init` method initializes the buffer with the track audio data stored in the given path[6]. Before starting playing the audio

[6] `Android` is an object defined in the actor's root lexical scope with helper functions for Android-specific facilities such as logging, loading the contents of a file stored in the device's SD card, etc.

Listing 5.5: The Song object prototype.

```
def Song := isolate: {
  def title; //string storing title of the song.
  def artist; // string storing artist of the song.
  def audioTrack; //object encapsulating the actual track.
  def init(artist, title, audioTrack) {
    self.artist := artist; self.title := title;
    self.audioTrack := audioTrack;
  };
  def ==(other) { (other.artist == artist).and: { other.title == title } };
  def play() { /*explained later*/ };
};
```

Listing 5.6: The audio track object prototype.

```
def audioTrack := lazyIsolate: {
  def buffer; // array of bytes containing the audio data.
  def track; // android.media.AudioTrack instance respresenting the audio resource.
  def init(audioFilePath) {
    buffer := Android.loadContentFromFile(audioFilePath);
  };
  def play() {
    if: (track == nil) then:{
      track := Android.createAudioTrack();
      track.write(buffer, 0, buffer.length());
    };
    track.play();
  };
};
```

track, the `play` method creates an `AudioTrack` instance and writes the audio data stored in the buffer to the audio hardware (if necessary). As such, the initialization of the `AudioTrack` Android object is delayed until it is necessary to play the song track.

Current distributed systems providing lazy parameter passing support three variants differing in when the copy of the object is acquired, either (1) when the reference is first accessed (called *implicit* or *on-access* lazy), (2) when explicitly requested in the method body (called *explicit* or *on-demand* lazy), or (3) immediately with the method invocation (called *imperative* lazy) [Eug03, LJE08]. Due to the event-loop concurrency model of AmbientTalk, an actor cannot be suspended in the middle of a computation. As such, we only provide explicit lazy semantics in which one can register a closure to be executed when the object copy is fetched from the remote peer. This is done using the **`when:fetched:`** control structure as shown in the code excerpt below.

```
def play(){
  when: audioTrack fetched: { |audioTrack|
   audioTrack.play();
  };
};
```

The code excerpt shows the implementation of the `play` method of the song object. It is called by the music player GUI when a song needs to be played. The closure passed to **`when:fetched:`** is applied to the copy of the audio track (once it is received). It calls the `play` method of the audio track object which actually makes the hardware resource play the song as previously explained. In the remainder of this section, we describe how we integrate lazy far references by means of our referencing framework.

### 5.2.4.1 Lazy reference data type

Lazy references are far references whose message reception and marshalling strategies deviate from that of normal far references. They basically apply three changes to the default semantics of far references. First, a lazy reference defines an additional meta method called `download` to be able to obtain a copy of the remote object. Second, asynchronous messages received by a lazy reference are forwarded to the remote object, or the local copy once downloaded. Finally, parameter passing of a lazy reference to a third party is altered to create a new lazy reference when the copy of the value is not downloaded.

Lazy references have been implemented as a custom transmitter-receptor pair by means of our referencing framework. Listing 5.7 shows the definition of the recep-

Listing 5.7: Receptor object for lazy references.

```
1  def makeLazyReceptor(target, targetDefaultMirror){
2    extend: defaultReceptor.new(target) with: {
3      def isDownloaded := false;
4      //default meta methods of a receptor
5      def accept(msg) {
6        if: (!isDownloaded) then:{
7          self.target <+ msg;
8        };
9      };
10     def unmarshallingStrategy(){
11       makeLazyTransmitter(self);
12     };
13     //method added to the receptor behaviour.
14     def download() {
15       isDownloaded := true;
16       targetDefaultMirror^pass();
17     }
18   }
19 };
```

tor for lazy references. The `makeLazyReceptor` constructor is parameterized with an isolate (playing the role of the target object of the lazy reference), and its mirror. Note that one can only create a lazy reference for pass-by-copy objects (isolates in AmbientTalk). The `isDownloaded` field encodes whether the isolate has already been transmitted. Initially, the isolate is not downloaded. When a `download` message is sent to the receptor, it returns a copy of the target object by triggering the default marshalling protocol for isolates. This happens by calling the `pass` method of the isolate's mirror (line 16). Once the copy is transmitted, the remote access to the target object is said to be terminated, and the receptor does not deliver messages to the target object anymore (line 6-8). When the target is first passed, a lazy receptor will be marshalled instead. Its unmarshalling strategy has been overridden to return a matching transmitter (lines 10-12).

Listing 5.8 shows the definition of the transmitter for lazy references. The `makeLazyTransmitter` constructor function is parameterized with the receptor associated to this transmitter. The transmitter also keeps a `isDownloaded` field to keep track whether the copy of the target object has been downloaded. Any asynchronous message received by the transmitter is either forwarded to the remote object if the copy is not yet downloaded, or forwarded to the copy if it is downloaded. This is done by overriding `schedule` (lines 7-13). The transmitter extends the default reference protocol with a `subscribe` method, which allows observers to be notified when the copy has been downloaded (lines 26-34). The addition of an observer makes the transmitter send the `download` message to the receptor which returns the copy of the target object (lines 31-41). The `download` message has been annotated with the `@ControlMessage` annotation to indicate that the receiver of the message is the receptor rather than the base-level target object. Once the copy is received, all subscribed observers are asynchronously notified with the downloaded value, and the transmitter is detached from the receptor (lines 35-39). The `detach` method is defined in the default transmitter and makes the interpreter remove the transmitter-receptor pair from the distributed data structures (explained later in Section 5.3.1.1).

Finally, the transmitter overrides the default marshalling protocol to apply pass-by-

Listing 5.8: Transmitter object for lazy references.

```
1  def makeLazyTransmitter(receptor){
2    extend: defaultTransmitter with:{
3      def isDownloaded := false;
4      def downloadedValue := nil;
5      def isDownloadInProgress := false;
6      def observers := [];
7      def schedule(msg){
8        if: isDownloaded then: {
9          downloadedValue <+ msg; // forward msg to the downloaded value;
10       } else:{
11         super^schedule(msg); // forward msg to the remote value;
12       }
13     };
14     def marshallingStrategy(){
15       if: isDownloaded then:{
16         downloadedValue;
17       } else:{
18         makeLazyTransmitter(self);
19       }
20     };
21     //methods added to the transmitter behaviour.
22     def subscribe(observer){
23       if: isDownloaded then:{
24         observer<-notifyDownloaded(downloadedValue);
25       } else:{
26         observers := observers + [observer];
27         // ask the receptor for a copy of the value.
28         if: !isDownloadInProgress then: { download() };
29       }
30     };
31     def download(){
32       isDownloadInProgress := true;
33       when: receptor<-download()@[ControlMessage, FutureMessage] becomes: {
34         |value|
35           super^detach();
36           isDownloaded := true;
37           downloadedValue := value;
38           observers.each:{ |obs| obs<-notifyDownloaded(downloadedValue);}
39       }
40     }
41   }
42 };
```

lazy-reference semantics when it is passed to third-party objects (lines 14-20). Figure 5.4 illustrates the marshalling of a lazy reference transmitter when the copy of the target object has been downloaded or not. It shows the object being marshalled in either case with a double line. If the copy has been downloaded when the transmitter is passed, the copy is returned so that its marshalling protocol creates a new receptor-transmitter pair representing a new lazy reference (as shown in Figure 5.4 (a)). If the copy was not downloaded when the transmitter is passed, a new lazy reference to the copy which still needs to be downloaded should be created. To this end, a new transmitter is created and passed to the third party (as shown in Figure 5.4 (b)). When the third-party object requires the copy to be downloaded (requested by means of the **when :fetched:** construct), the `download` method of the new transmitter (T') will call the `download` method on the transmitter (T) corresponding to the original lazy reference. This is possible because `download` returns a future and a future chain is created between the transmitters handed out from the original lazy reference. When the original transmitter obtains the copy, the transmitters handed out to third party objects will be notified with the copy similarly to other observers.

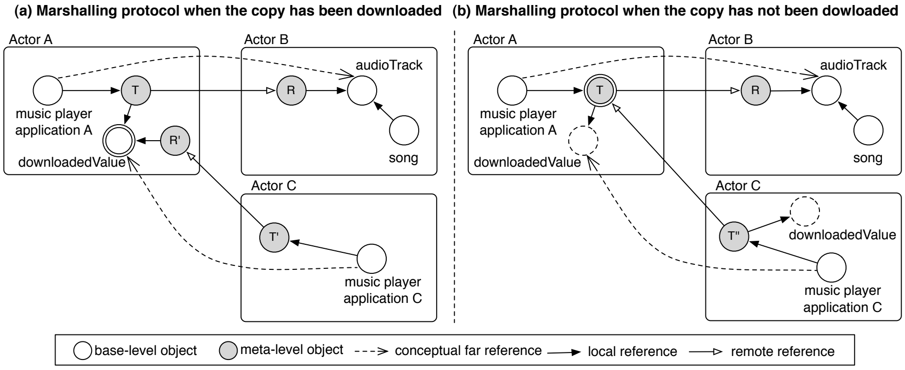

(a) Marshalling protocol when the copy has been downloaded
(b) Marshalling protocol when the copy has not been dowloaded
Actor A
Actor B
Actor C
audioTrack
T
R
R'
T'
T''
music player application A
downloadedValue
song
music player application C
base-level object
meta-level object
conceptual far reference
local reference
remote reference

Figure 5.4: Marshalling protocol of lazy references

#### 5.2.4.2 Integration in the Object Marshalling Protocol

Now that we explained the implementation of a lazy reference, we turn our attention to the language constructs provided to the programer to obtain and use lazy far references. The following code excerpt shows the implementation of `lazyIsolate:` construct.

```
def lazyIsolate: closure {
  isolate: closure mirroredBy: ( mirror: {
    def pass() {
      makeLazyReceptor(self.base, super); // apply pass-by-lazy-reference semantics.
    };
  });
};
```

The `lazyIsolate:` construct creates an isolate with the given closure (containing the code with which to initialize the object and the new object's lexical parent) and alters its marshalling protocol to integrate pass-by-lazy-reference semantics. This is done by overriding the isolate's meta-level `pass` method. The method is specialized such that it returns a receptor of a lazy reference, rather than applying the default pass-by-copy semantics (i.e., returning a copy of the base-level isolate). The marshalling of the base-level isolate is thus delegated to the receptor.

The reader may have noted that the custom receptor for lazy references is not installed using the **`becomeReferencedBy:`** meta operation. This is because calling it on an isolate turns it into a (pass-by-reference) object. Similarly to implicit mirrors on actors, different referencing strategies can then be installed at any time during the lifetime of the object by means of **`becomeReferencedBy:`**. However, once a receptor is installed, it is not possible to uninstall it, i.e., turn the object back into an isolate. Allowing pass-by-reference objects to be passed by copy can lead to several conceptual problems with respect to scoping issues and object identity (that we discuss in Section 5.5). As such, the developer needs to manually pass the isolate's mirror to the receptor of the lazy references so that the default pass-by-copy semantics can be invoked at a later stage. In a sense, lazy parameter passing can be considered as an exceptional referencing abstraction since it is built for isolates, rather than for objects.

Below is the definition of the **`when:fetched:`** control structure that allows base-level programmers to initiate the transfer of the copy of the isolate.

```
def when: lazyFarReference fetched: closure {
  (reflectOnReference: lazyFarReference).subscribe( object: {
    def notifyDownloaded(value) { closure(value) }
  });
};
```

The `subscribe` method has been previously explained as part of the transmitter of a lazy reference. Recall that a transmitter is a meta-level object representing the source of a far reference. **`reflectOnReference:`** acquires the transmitter from the reference data type allowing the language construct to invoke the method at the meta-level rather than on the base-level far reference object itself.

## 5.3 Reflection on Actors

One of the novelties of AmbientTalk's reflective layer is that it applies mirror-based reflection to actors. As explained in the previous chapter, AmbientTalk's meta-actor protocol is encapsulated in the actor mirror which allows one to reflect on the event loop executing a (meta)program. As mentioned in Table 4.1, the meta-actor protocol provides hooks to influence asynchronous message sending and reception, the service discovery protocol, the object mirror creation, and the reference creation. In this section, we discuss the changes and extensions made to the actor mirror in AmbientTalk/M for easing the development of referencing, and failure handling abstractions.

### 5.3.1 Extensions to the Meta-Actor Protocol

In order to accommodate the new mechanism to build referencing abstractions explained in the previous section, we adapted the reference creation protocol. It consist of the `createReference` method invoked whenever a pass-by-reference object is parameter passed between actors. In our approach, `createReference` returns the default receptor instead of a far reference data type. This allows metaprogrammers to define their own transmitter-receptor pair to *all* references exported to other actors upon parameter passing objects. As previously mentioned, the default transmitter-receptor pair encodes a far reference.

In addition, we added to the meta-actor protocol hooks to influence the reference management protocol, and the method invocation protocol on objects. Table 5.4 provides an overview of the operations added to explicit mirrors on actors[7]. In the remainder of this section we briefly discuss both protocols.

#### 5.3.1.1 Reference Management Protocol

The reference management protocol reifies the distributed structure of an actor as a table of object references. Each actor maintains an *object reference table*, reminiscent of the object table in network objects [BNOW93]. The object reference table contains two different kinds of mappings. First, a mapping between identifiers and receptors to local objects (if there is one). The `addReceptor` method is called to enter a receptor into the actor's object reference table when the local object is first parameter passed to an actor. It remains there until it is removed implicitly by the interpreter when it detects the deletion of all its associated transmitters, or it is explicitly removed by programmers. Removing a receptor (in either way) happens by means of the `removeReceptor` method of the recipient actor's mirror.

[7] Note that methods defined by explicit mirrors are also exposed by implicit mirrors.

Table 5.4: Additional methods in the API of explicit mirrors on actors.

| | |
|---|---|
| *Reference management protocol* | |
| addTransmitter(transmitter) | Adds a new transmitter to the object table. |
| removeTransmitter(transmitter) | Removes a transmitter from the object table. |
| grabTransmitters(reference) | Returns an array of all transmitter to a remote object. |
| addReceptor(receptor) | Adds a new receptor to the object table. |
| removeReceptor(receptor) | Removes a receptor from the object table. |
| grabReceptors(toObject) | Returns an array of all receptors to a local object. |
| listReceptors() | Returns an array of all receptors to local objects. |
| listTransmitters() | Returns an array of all transmitters to remote objects. |
| *Method invocation protocol* | |
| addBeforeMethodInvocationObserver(<br>closure,object,selector,filter) | Registers a closure as an event handler triggered before a matching method is invoked on an object. |
| addAfterMethodInvocationObserver(<br>closure,object,selector,filter) | Registers a closure as an event handler triggered after a matching method is invoked on an object. |

Second, the object reference table contains entries for all transmitters that exist in the actor. In this case, it maps identifiers (of service objects) to the local transmitter for a service object. The `addTransmitter` method is called when the actor first acquires a far reference to the service object as a result of either a discovery request or the reception of an asynchronous message carrying an object as argument. Similarly to network objects, the object reference table holds a *weak reference* to the transmitter, which allows the underlying local garbage collector to collect it when there is no other reference to it except through the table. The `removeTransmitter` method removes a transmitter from the table of the recipient actor's mirror and notifies the corresponding receptor(s) of the removed transmitter.

By default, as previously mentioned, there exists only one transmitter and one receptor per service object in each actor. However, meta programmers can override these semantics, e.g., to provide a different transmitter -receptor pair per far reference handed out, resulting in several receptors in the object reference table for a service object. The `grabReceptors` method returns an array of all receptors for a given service object. Similarly, the method `grabTransmitters` returns an array of all transmitter for a service object. In this case, the method extracts the identifier of the service object through the given reference data type, and returns all transmitters associated with the same object identifier. Although referencing abstractions that require several transmitters for a service object are not common, this hook supports the development of referencing abstractions that integrate subject-oriented programming [HO93] in which the client object of the reference is involved to ascertain the kind of access that can be granted. Finally, `listTransmitters` and `listReceptors` returns an array of, respectively, all transmitters and receptors in the actor's object table. As in the case of `listOutgoingLetters`, these arrays are not causally connected to the implementation-level data structures: they just provide a snapshot of the actor's object table.

Listing 5.9: Definition of a method invocation observer.

```
def dictionaryObject := {
  def put(entry) {...};
  def get(entry) {...};
};
before: dictionaryObject invokes: `put do: {|entry|
  log(" added " + entry + " to dictionary");
};
```

The object reference table plays a crucial role in the distributed behaviour of an actor. First, it is used to find the service object referred to by an incoming asynchronous message, and to perform the marshalling and unmarshalling of a service object transmitted (by service discovery or parameter passing). Second, the table is central to distributed garbage collection and its interaction with the local garbage collector. The service objects targeted by the transmitters and receptor objects stored in the object reference table are considered as *root objects* for the local garbage collector. As such, the hooks provided are crucial to build high level distributed memory management abstractions within the language itself. Finally, it allows to create constructs to simulate network failures as we describe in 5.3.2.

#### 5.3.1.2 Method Invocation Protocol

We introduce the method invocation protocol in order to deal with the fact that implicit mirrors cannot be dynamically installed on objects (cf. Section 5.1). The goal of this protocol is to make the interpreter notify a metaprogram of the execution of a method on an object, while preventing the metaprogram from changing the semantics of the executed method as this may cause unwanted interference with implicit mirrors installed on the observed object. As shown in table 5.4, the protocol consists of two methods that register an *observer* to be notified before and after the execution of a method invocation on a given object. They are called `addBeforeMethodInvocationObserver` and `addAfterMethodInvocationObserver`, respectively. These methods are parameterized with a closure that subscribes to the object to observe. The closure is triggered whenever the object executes a method whose name matches a selector and a filter (a predicate on the actual method invocation).

For syntactic convenience, we feature keyworded syntax constructs for those meta methods in a similar way to other AmbientTalk language constructs such as **`whenever :discovered:`**. As a concrete usage example, consider listing 5.9 that captures all the entry additions to a dictionary object. The **`before:invokes:do:`** function installs a before method observer for the `put` method on the `dictionaryObject`. As mentioned in Section 4.6.1, the backquote denotes a symbol. The **`do:`** closure is triggered on each method invocation whose selector is `put`. It receives as arguments the actual parameters of the method being observed, in this case, `put` has only one parameter named `entry`. The order of the arguments on the **`do:`** closure is expected to be the same as the observed method. Analogously, an after observer can be registered using the **`after:invokes:do:`** function. Both functions have an extended version with a predicate parameter to filter method invocations based on their actual parameters. The code snippet below shows an equivalent extended version for the example given in listing 5.9.

```
before: dictionaryObject invokes: `put with: {|entry| true} do: {|entry, result|
  log(" added " + entry + " to dictionary");
};
```

The **with:** closure represents the filter applied to the `put` method invocation. In this case, the filter always returns true. The **with:** closure arguments correspond to the actual arguments of the method to observe. Similarly to the **do:** closure, the order of the arguments on the closure implementing the filter is expected to be the same as the observed operation (`put`).

The `addBeforeMethodInvocationObserver` and `addAfterMethodInvocationObserver` methods register a closure as an *event handler* for a method invocation, i.e., the code of the closure is always delayed. The closure will thus be asynchronously triggered whenever a matching method invocation is observed. More concretely, when a matching method invocation is observed, the interpreter sends a message of the form `closure←apply(args)@ObserverMessage`. As we mentioned in Section 5.2.2.3, observer messages are executed as regular asynchronous messages, making sure that the closure is applied serially with respect to other executions in the actor.

It is important to remark that these operations can be used to observe method invocations on both base-level or meta-level objects. In AmbientTalk a (mirror) meta-level object can be seen as a regular object that provides reflective access to a base-level object. As such, a mirror can also be mirrored by another mirror, and so on. As a concrete example, consider again the example of a debugger given in Section 5.1. The code snippet below shows how a debugger can update its GUI whenever an assignment changes the value of an object's field by means of the **after:** function.

```
after: (reflect: objectInDebug) invokes: `invoke with: {|delegate, invocation|
  invocation.selector.isAssignmentSymbol();
}  do: { |delegate, invocation, result|
  // result is bound to the value being assigned.
  debuggerGUI<-updateField(objectInDebug, invocation.selector, result);
};
```

In this case, we install an after observer on the meta-level method invocation of `invoke` with a filter to trap assignments. This means that the object to be observed is the mirror of a base-level object participating in the debugging session (obtained with the **reflect:** operation). **with:** closure defines a filter that takes as parameter two arguments as specified by the `invoke` method signatures. As mentioned before, the `isAssignmentSymbol` helper function checks whether the selector of the method invocation corresponds to an assignment symbol. The body of the **do:** closure sends an asynchronous message to the GUI to update the the information displayed for the `objectInDebug` object. When registering an after observer, the result of the method invocation is also passed as argument.[8] The `updateField` messages sends to the GUI the selector of the field name being changed, and the new value obtained through the result argument of the `invoke` method.

### 5.3.2 Making Entities Fail

Causing a partial failure programmatically can be useful for a number of reasons. First, it allows to isolate certain actors (which are e.g., malicious, spamming, suffering security attacks, etc.) from the rest of the network. Secondly, it forms the basis to build meta programs that monitor or simulate the distributed behaviour of an application and its resilience to partial failures. This argument is valid for any distributed system, but

[8] By convention, the result will be always the last argument of the closure.

in a MANET setting it becomes crucial to understand the distributed behaviour of applications and detect misbehaviours at early development stages as they are exposed to a highly dynamic environment. We propose to provide support to make language entities fail based on the reference management protocol described in Section 5.3.1.1.

As previously explained, AmbientTalk provides the `takeOffline:` primitive that allows us to disable all far references to a service object. AmbientTalk/M allows to implement the `takeOffline:` primitive reflectively using the reference management protocol. The method first looks up all the receptors for a given service object, and then calls `removeReceptor` on each. In addition, we also provide a `takeActorOffline` function which disables all incoming and outgoing references to objects in that actor.

An important use of `takeActorOffline` is unit testing. AmbientTalk features a unit testing framework that provides support to test the distributed behaviour of an application exploiting its concurrency model. Developers can define methods prefixed with `testAsync` which return a future. The framework only processes subsequent tests once this future is resolved or ruined. In many MANET applications, a recurring pattern is to take offline the object representing the remote interface of the local application to other applications to trigger the failure handling code of the remote applications. To do so, developers typically use the `takeOffline:` primitive but, it does not prevent the local application to send messages to the remote applications. The `takeActorOffline()` function prevents programmers from writing boilerplate code for the bookkeeping of the references to remote applications.

AmbientTalk/M also allows to reflectively implement the `disconnect:` primitive that logically disconnect an object it receives as an argument. The primitive returns an object whose `reconnect` method reestablishes the object's connection. In this case, the implementation first removes the transmitter or receptor for the given object from the object reference table, which is re-added upon calling the `reconnect`.

## 5.4 Discussion

We have described the new meta-level engineering support that AmbientTalk/M introduces to the existing AmbientTalk meta-level architecture. First, it represents far references as an open framework in which developers can build referencing abstractions in a high-level and customisable way. Developers can monitor and redefine the distributed concerns of an object *solely* by means of custom transmitter-receptor pairs, abstracting distribution in dedicated metaobjects which are cleanly separated from application-level code. Second, it revisits AmbientTalk's meta-actor protocol and introduces an observer mechanism that allows developers to react to changes on objects made by the interpreter without requiring the installation of implicit mirrors. We now motivate the more important design decisions of AmbientTalk/M's extensions and highlight a number of techniques which have influenced the design of AmbientTalk/M.

**First-class far references.** Our design of the transmitter-receptor model has been driven by our experiences with AmbientTalk in which references are expressed as mirages, and the analysis of several referencing abstractions built in the language including futures, ambient references, leased references, lazy references[9], and network-aware references [PHGD11]. One important idea that shaped the design of our architecture is that we consider that a reference provides a way to access a remote object and get

[9] The first implementation of lazy references was conducted in Cardozo's master thesis [Alv09]

certain rights to it. As we target MANET applications, one cannot rely on the presence of infrastructure acting as mediator or coordinator of a distributed interaction. As such, remote references become the abstraction to both designate remote objects and control (and even limit) how objects can be accessed. This also motivated the explicit representation in our architecture of both sides of a remote interaction by introducing the concept of transmitters and receptors.

Similarly to implicit mirrors on actors, receptors can be installed at any time during the lifetime of a service object (by means of **`becomeReferencedBy:`**). This design is mainly motivated by our goal of separating application functionality as much as possible from distribution concerns. In particular, the transmitter-receptor model aims to decouple the functionality provided by an object from the way how remote objects can access such functionality. As a result, developers can decide parameter passing semantics at runtime independently of the object type (determined when the object is defined). Moreover, this design is flexible enough to allow different referencing strategies to be handed out to different client objects, for example, to introduce subject-oriented programming or role-based models [KO96] into asynchronous method invocations. Note, however, that we still provide a **`object:referencedBy:`** construct to define a default referencing strategy for an object upon its creation.

Important to remark is that transmitters reintroduce an explicit representation of the *outbox* and *sentbox* of an actor (present in AmbientTalk/1), i.e., the mailbox which reifies outgoing and sent messages to other actors, respectively. They can be obtained by overriding `serve` or `leave` meta methods of a transmitter, respectively. Similarly to AmbientTalk, each reference contains a part of the outbox or sentbox, but in AmbientTalk/M the meta-actor protocol now provides means to access all transmitters in an actor. As such, it is easier to monitor or alter the communication traces of an actor.

To conclude, transmitters and receptors have been specially designed to model asynchronous communication patterns between objects and provide control over the messages sent in their interactions. This allow developers to build referencing abstractions which require both senders and receivers of messages to intercept asynchronous messages, and control which parties may receive a reference. Although our architecture naturally supports point-to-point communication abstractions, it is flexible enough to build abstractions based on other communication patterns such as one-to-many patterns as shown with the prototype implementation of omnireferences in listing 5.4.

**Meta-Actor Protocol Extensions.** The addition of the observer mechanism into the actor meta protocol is mainly motivated by our experiences building tool support for ambient-oriented programs. As discussed in Section 5.1.3, implicit mirrors on objects provide the appropriate level of granularity to build certain tools such as inspectors and tracers, but they are too inflexible to allow those metaprograms to be dynamically enabled and disabled at runtime on objects (which do not define an implicit mirror). To overcome that issue, we introduce the method invocation protocol into actors in which developers can register observers to be notified before or after the execution of a method invocation without actually having to override those MOP methods. This allows developers to use the same API to observe the execution of a base-level, and a meta-level method invocation. Since mirrors are often used to encode language features, certain metaprograms (such as debuggers) would like to observe method invocations of meta-level objects rather than base-level objects. Introducing this protocol directly on the explicit mirror it would require a way to distinguish whether the observer is installed on the mirror's reflectee or the mirror itself. With our current implementation, the distinction is made directly in the API which can be called passing as argument either a base-level object or a mirror object.

The method invocation protocol may be reminiscent of the before/after pattern present in aspect-oriented programming [KLM$^+$97] and languages like Lisp or CLOS. However, there are two differences as how it is implemented. First, one is registering observers before/after a method invocation, rather than computation to be executed before/after a method invocation. This means that the execution of the observed method invocation is not intercepted. This is important because otherwise one could cause unwanted interference with implicit mirrors installed on an object. The registered observers may still perform operations that changes the state of the observed object, but they are always executed after the observed method invocation (because observers are notified asynchronously). Second, we do not provide around methods since it would imply that the observers could determine when, how and even if the method invocation is actually executed. Similarly to some AOP languages, our after method has access to the return value of the observed method invocation (but it cannot modify it).

## 5.5 Limitations and Future Work

In this section, we discuss limitations of our reflective architecture and give some directions for future work.

**Composing Referencing Abstractions.** The default transmitter-receptor pair defines the behaviour common to remote object references operating in an asynchronous communication model. In this chapter we have discussed how developers can create variations on the default behaviour. Several referencing abstractions, however, may share similar behaviour for a particular protocol or subset of a protocol (e.g., they may react similarly to disconnections). We rely on AmbientTalk's traits to combine existing transmitter-receptor pairs into new ones. A trait is an object module that provides a set of methods and requires a set of methods. When a trait is imported into a *composite* object, the trait's provided methods are "copy-pasted" to the composite, and the composite must implement all the trait's required methods. Different variations of the protocols described for transmitter-receptor pairs can be modularized into traits allowing them to be reused in new transmitter-receptor pairs at runtime. Composing overlapping protocols require developer intervention, but AmbientTalk's trait composition mechanism helps to detect where the protocols collide. Developers must then resolve possible name clashes by excluding or aliasing names in the composite object. As future work, we aim to standardize all distributed abstractions in AmbientTalk's standard library into different traits using the API exposed by transmitters and receptors. We would also like to explore a mechanism that guides trait composition to avoid the composition of *semantically incompatible* protocols, i.e., the protocol composition does not raise a name clash but the resulting behaviour is not correct.

**Parameter Passing Framework.** Our architecture allows a programmer to dynamically install different referencing strategies for an object at runtime. This means developers can decide on parameter passing semantics for a pass-by-reference object independently of its object type. However, such flexibility does not extend to isolates (AmbientTalk objects that are parameter passed by copy). Allowing developers to decide at runtime whether an object is passed by-reference or by-copy is problematic. First, the notion of object identity is different in objects and isolates. Since different versions of the same isolate may coexist within the system, an isolate has multiple identities. In addition, the notion of scoping is also different. Since isolates do not have access to their enclosing lexical scope, they need to explicitly import in their object scope any object it needs to call (e.g., library functions). Introducing an automatic algorithm to determine the bindings required by an isolate may lead to very subtle bugs as the programmer may not be aware of which code is copied along with the isolate. Finally, since AmbientTalk features different operators for local or remote message sending, it forces developers to distinguish upfront isolates from objects.

Despite those issues, it is our vision that MANETs require a more flexible parameter passing model with a richer set of parameter passing semantics that suits the different kinds of interactions among objects. A first experiment to revisit object parameter passing for an asynchronous communication model has been conducted in Cardozo's master thesis [Alv09]. He observes that a way to bridge the gap between pass-by-copy and pass-by-reference semantics in ambient-oriented systems is by making object transfers more flexible in two aspects: when to transfer an object, and what part of the object graph to transfer. He consequently proposed the Proteus model which introduces ideas of lazy parameter passing (cf. Section 5.2.4) and adaptive techniques [Lop96] into AmbientTalk's asynchronous distributed model. An interesting topic of future work would be to study whether and how to express different parameter passing semantics for pass-by-copy objects using the transmitter and receptor meta objects. Although our far references API is flexible enough to implement replication mechanisms similar to the ones in GARF [GGM94] by overriding the reference marshalling and asynchronous message process protocols, it remains to be explored whether that API is robust enough to build other semantics such as pass-by-copy-restore [TS03], caching [ET01] and streaming [YCC$^+$06].

**Security.** Although the receptor-transmitter model does not offer any built-in security guarantees, it provides developers with the appropriate infrastructure to build abstractions that impose certain security properties. For example, receptors are the ideal place for implementing security policy enforcement. They naturally support security mechanisms derived from role-based principals [RK98a]. The basic concept in this model is a *security metaobject* which is attached to an object reference and controls all the calls via that reference. Since meta objects are attached on a per-reference basis, one can define different access privileges when interacting with different parties. The model also provide means to automatically attach security metaobjects when references are exchanged among different parties.

In his dissertation [Van08], Van Cutsem already discussed initial ideas about how to reconcile AmbientTalk with object-capability security à la E [MMF00, Mil06]. The object-capability model also seems to us a suitable model for securing distributed interactions in AmbientTalk/M because it uses the object reference graph as the access control graph. The main ideas are that objects can be *solely* affected by sending messages through a reference, and that there exists a well-defined set of operations to acquire an object reference ( i.e., object creation, and passing it as parameter in a message send). Holding a reference to another object thus implies *authority* to manipulate it. In contrast to role-based principals, such a model does not perform access control checks at the receiver end when the reference is used. Rather, its goal is to avoid excessive authority which may lead to abuse (e.g., by malicious objects) by limiting the spread of object references.

Combining object-capability security with the receptor-transmitter model may raise some issues due to the use of reflection. When reflecting upon a reference data type, client objects gain access to the transmitter of the reference (rather than the remote object targeted by the reference). Although this does not convey more power in the sense of acquiring an object reference to the target object or the target's mirror, it enables client objects to gain control over the distribution of access rights. As a result, client objects can modify how an object reference is spread to third party objects, and possibly leak authority to untrusted parties. Several works studied the security issues raised by reflection within the context of MOPs [CV01, CHV01, WS99]. Nevertheless, it is necessary to investigate how those works apply to a mirror-based reflective architecture for distribution. Therefore, building a suitable security mechanism on top of the receptor-transmitter model is an important topic for future research.

We conclude our discussion about AmbientTalk/M meta-level engineering with a note on performance. Like AmbientTalk, AmbientTalk/M has not been designed an an optimized software development platform, rather as a research artifact used as a language laboratory. Assessing the performance impact of the new meta-level abstractions remains an open issue for future work. Following the principles of *partial behavioural reflection* [TNCC03], there are a number of optimisations that could be introduced in AmbientTalk/M. We have already limited the reification of some meta-level operations to object references with custom receptors in the current implementation. This allows us to disable the asynchronous message process protocol when a default receptor is used, avoiding unnecessary interception of method invocations resulting from an asynchronous message send. Such an optimization is called *entity selection*, and it has also been applied to mirages and ordinary objects in AmbientTalk.

## 5.6 Notes on Related Work

We now discuss prior work on reflective meta-level architectures for distributed communication and actors.

**Reification of Distributed Communication** Reflecting both ends of a remote interaction has been previously explored in a few reflective architectures for traditional distributed systems. CodA [McA95] was pioneer in providing a dedicated MOP to ease the development of distributed objects, especially the communication aspect. It provides a relatively fine-grained meta-level architecture in which a base-level object is controlled by a set of seven metaobjects. The major drawback of CodA is that a metaobject can only monitor a message once it has been received by the target object. As a result, the metaobject cannot dispatch messages based on the sender's identity. For example, the metaobject responsible for accepting a message (called Accept) does not know if the sender is a distributed or local object. This information is relevant to e.g., perform access control checks when objects are remotely accessed.

Our view of reifying each communication endpoint separately is based on the mailer/encapsulator model of GARF [GGM94]. Encapsulators wrap data objects by controlling how they send and receive messages, while mailers perform communication between encapsulators. The asynchronous message process protocol on receptors and transmitters is based on the same idea. Encapsulators and mailers represents the root of the class hierarchy for the basic communication model. Support for asynchronous communications, persistence and replication have been built-in as subclasses. Although encapsulators and mailers are meta-level objects, they are not completely stratified in

GARF; developers need to override a (data) object constructor to bind it to an encapsulator and to a mailer. In contrast, in our architecture receptors are installed by means of a dedicated operation separated from base-level code and transmitters are also automatically instantiated by the interpreter upon the deserialization of an encapsulator.

Instead of reifying communication based on a metaobject model, the channel reification model proposed to reify the communication channel between a sender and a receiver in a separate meta-object called a *channel* [ACDG98]. A channel acts as a communication manager that traps and modify each message sent to a receiver. The model was then extended to support one-to-many communication giving rise to the *multi-channel reification* model [Caz01]. The main advantage of this approach in comparison to CodA and GARF is that developers can filter messages before they are delivered to the target object. Each multi-channel is characterized by its behaviour (the kind of communication semantics it defines) and the set of receiver objects. This allows that among the same group of senders and receivers can exchange different kinds of messages. However, it requires to describe the set of receivers of a multi-channel by explicitly enumerating the receivers when the channel is defined. This makes the approach unsuitable for distributed systems in which the number of participants in an interaction may not be known beforehand ( like in a MANET).

In the context of RPC, some platforms such as CORBA and DeXteR [GTKT08] use the notion of an Interceptor object to expose the invocation context of a remote call. DeXteR is a framework designed to express different parameter passing semantics built on top of Java RMI. Like our first-class references, DeXteR provides interceptor points on both client and server sites of a remote call. Those points enable developers to modify the original invocation arguments for one remote call at runtime. They are equivalent to overriding the `schedule` and `accept` methods of our API and modifying the message arguments. DeXteR enables also sending custom information between client and the server-side of an interceptor by piggy-backing it to the original invocation context. This corresponds in AmbientTalk/M to send an asynchronous message annotated with the `@ControlMessage` annotation between an transmitter and a receptor. Although flexible, due to its reliance on Java RMI, it is difficult to build referencing abstractions which deviate from a point-to-point synchronous communication model. In addition, the system brings unnecessary additional complexity to the generation of annotations to introduce for a new parameter passing semantics.

In general, current reflective models for distributed computing are often limited to trapping messages sent to an object and implement the behaviour of that invocation, lacking the reification of important aspects such as object marshalling. Since most of them were designed for stationary networks, they also do not provide support for the specific requirements of MANETs such as exposing the environmental context of distributed communications, and supporting arity-decoupling communication (the fact that some messages may have to be dispatched to several destinations).

**Reification of Actors** Many classic actor frameworks and languages provide the notion of a *meta-actor* [DA07]. Typically meta-actors only reify the message passing protocol, allowing developers to intercept incoming and outgoing messages to alter the behaviour of the actor system. Interestingly, in the Two-level actor model (TLAM), a formal model to reason about the actor-based interactions, a reachability snapshot can be created using meta-actors [VT95]. However, those reflective architectures are built for *active objects*, the traditional representation of actors. As previously mentioned, AmbientTalk's reflection on actors is already innovative on its own as it combines a mirror-based reflective architecture into an event loop concurrency model. To the best of our knowledge, the additions to manage references and reify the object reference table are not found in meta-actor architectures. However, certain object-oriented frameworks like OpenTalk provide a limited interface to manage the object table. In OpenTalk, object adaptors can keep a record of the object references they export by means of ObjectTables class which provides three methods to add and get entries.

## 5.7 Conclusion

In this chapter, we presented AmbientTalk/M, a variant of AmbientTalk which enhances its meta-level engineering with support for building custom referencing and failure handling abstractions, and support for reacting to the manipulation of objects by the interpreter by dynamically adding observers on explicit mirrors. The reflective capabilities that we introduced in AmbientTalk/M play a vital role in the remainder of this dissertation. First, first-class references provide an appropriate platform on top of which we can build language abstractions to deal with the effects of partial failures *on both ends* of a communication abstraction. Second, the revisited meta-level architecture allows us to easily build metaprograms that monitor and control the distributed behaviour of actors within AmbientTalk itself. In particular, we will employ it to build a debugger for AmbientTalk programs in the second part of this dissertation.

On the whole, the abstractions introduced in Ambientalk/M facilitate our experiments on language constructs for dealing with partial failures, and allow them to be reused for building software development tools for ambient-oriented application. Having introduced both AmbientTalk/2 and the reflective architecture provided by AmbientTalk/M, the next chapter describes our failure handling model, and its integration as a custom object referencing abstraction.

We conclude this chapter with a note on terminology: for simplicity's sake, we will use the term AmbientTalk to refer to AmbientTalk/M in the remainder of this dissertation. The languages' full names will be used to clarify the text when necessary.

# Chapter 6

# Ambient-Oriented Leasing

In this dissertation, we go one step further in distribution support for MANET applications by introducing abstractions to deal with a number of the failure handling criteria exposed in Section 2.4. As we already explained in Section 2.3, it is impossible to accurately distinguish a transient from a permanent failure in a MANET. To deal with the resulting uncertainty, developers need to make assumptions about the timing behaviour of the application. To this end, we explore the concept of *leasing* as a programming abstraction that allows developers to capture timing assumptions and react accordingly. However, as a result of the hardware characteristics of mobile networks (cf. Section 2.1), a leasing mechanism needs to operate under assumptions different from traditional distributed systems.

We start this chapter by describing four criteria that a leasing model needs to exhibit to in order to be usable in a MANET. We then describe the notion of a lease that should be integrated into a software platform designed to build applications running on MANETs. With our notion of lease introduced, we define two novel language abstractions: *leased object references*, which enable decoupled communication with remote parties while tolerating both transient and permanent failures, and *due-type messages*, which enable computation to be processed within time-based guarantees. We also study some scoping language constructs to reduce the programming effort introduced by leased object references and due-type messages. Finally, we describe how leased object references are offered as an extensive framework in which many leasing patterns can be expressed. This allows advanced programmers to express custom leased-based interaction patterns among distributed processes according to the needs of their application.

## 6.1 Motivation

In this section, we motivate the main design choices of a leasing model for an ambient-oriented programming model. We first describe a scenario - that we use as running example throughout this chapter - and subsequently we use this scenario to derive a number of criteria for a leasing model to adhere to for the failure handling model criteria postulated in Section 2.4.

### 6.1.1 Scenario: the Mobile Music Player

Consider a music player running on mobile devices. The music player contains a library of songs. When two people using the music player enter one another's personal area network, the music players set up an ad hoc network and exchange their music library's list (not necessarily the songs themselves). After the exchange, the music player can calculate the percentage of songs both users have in common. If this percentage exceeds a certain threshold, the music player can e.g., inform the user that someone with a similar taste in music is nearby.

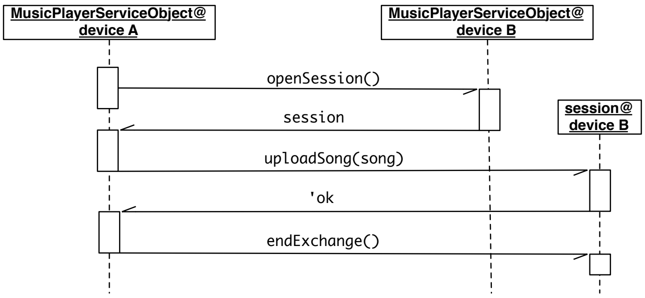


Figure 6.1: The music library exchange protocol

Figure 6.1 gives a graphical overview of the music library exchange protocol modeled in an asynchronous distributed object-oriented system. The figure depicts the stages of the protocol from the point of view of the music player on device `A`. This protocol is executed simultaneously on both devices. Once both devices have discovered each other, the music player running on `A` asks the remote peer `B` to start a session to exchange its library index by sending an `openSession` message. In response to it, the remote peer returns a new `session` object which implements methods that allow the remote music player to send song information (`uploadSong`) and to signal the end of the library exchange (`endExchange`).

When implementing the music library exchange protocol, we should take care of network failures. First, the application should remain responsive and cope with temporary failures, e.g., the application may want to inform the end-user that the transmission of songs is temporary stopped. However, the application must also cope with permanent failures. The main problem is that if a peer disconnects in the middle of the library exchange, the session will never terminate, and the application will consume unnecessary resources, e.g., the partially uploaded library of the device `A`.

### 6.1.2 Criteria for a Leasing Model in MANETs

Leasing provides a solution to deal with the effects engendered by partial failures based on the temporal availability of resources [GC89]. A lease is a contract that gives the right to access a resource for a specific duration that is negotiated between the owner of a resource ( called the *lease holder*) and a resource user (called the *lease grantor*). Once a lease is granted, both lease grantor and holder know the period of time that the resource is available (called the *lease term*) and conditions under which the lease is valid (called the *lease statement*).

In a MANET, leasing needs to operate under assumptions different from classical distributed systems, as a result of the hardware characteristics of the network (cf. Section 2.1). Below, we describe four criteria that need to be fulfilled by a leasing model to be usable in a MANET.

#### 6.1.2.1 Criterion #1: Leasing an Intermittent Connection

In general, a lease denotes a temporal restriction on the logical connection between lease holder and grantor. At the software level, a logical connection is represented by a communication link. Because of the volatile connection phenomenon exhibited by MANETs, communication links are often intermittent. However, that does not imply that the logical connection should be terminated. In our running example, once the service objects that represent the music player application have discovered one another, they need to set up a session to exchange their music libraries. Such a session should be *leased*, such that both music players can gracefully terminate their interaction in the face of a persistent disconnection. However, if a user temporarily moves out of range, the resulting transient disconnection should not immediately cause the exchange to fail since the user may reconnect shortly. A leasing model for MANETs must take this into account: the disconnection of a device does not indicate that resources associated with the logical connection can already be cleared, since the communication link may be restored later.

#### 6.1.2.2 Criterion #2: Leasing Management Patterns

As we will see, the advantage of leasing is that it allows both lease grantor and holder to distinguish a transient from a permanent failure by approximating permanent failures as disconnections that exceed the lease term. However, leasing can only provide an approximation of when a disconnection is permanent. The quality of the approximation depends on the accuracy of the selected lease period. *Selecting a suitable lease period is not straightforward and it requires developers to understand the behaviour of mobile devices in the physical world and know factors such as the frequency of disconnections and reconnections.* Any leasing model has to deal with this issue, but this issue is exacerbated in MANETs due to the unpredictable mobility of mobile devices.

In order to tackle this issue, a leasing model for MANETs should provide support to aid the developer determine the lifetime of leasing agreements. In our running example, devices collaborate in the exchange of their music library indexes. As long as the exchange is active, i.e., `uploadSong` messages are received, the session should remain active. The session could thus be exported using a lease which is automatically extended each time it receives a message. The lease should then be revoked either explicitly when a client sends the `endExchange` message to indicate the end of the library exchange, or implicitly if the lease time has elapsed. Other collaborations may involve objects adhering to a *single call* pattern in which a lease is automatically revoked upon the reception of one single message. For example, this pattern is exhibited by futures. Recall from Section 4.3.2 that futures are objects passed along with a message in order for the receiver of the message to be able to return a value . Futures are accessed only *once* by the receiver of a message to deliver the computed return value or an exception (if the message invocation failed). Hence, the need for the single call pattern.

To sum up, a desirable characteristic of a leasing model for MANETs is the incorporation of a number of built-in leasing patterns that allow programmer to select the most appropriate leasing agreement for his distributed interactions.

#### 6.1.2.3 Criterion #3: A Customizable Leasing Framework

Due to the openness and heterogeneity of mobile ad hoc networks, applications interact with a very diverse range of mobile devices. Of course, as different kinds of collaboration can be set up, various kinds of leasing agreements are definitely possible. Providing only a fixed set of leasing patterns is not desirable, because it is hard to predict the leasing agreement that fits best for each MANET application. Therefore, a leasing model for MANETs should also be equipped with facilities to enable experienced developers to construct new and tailored leasing variants.

#### 6.1.2.4 Criterion #4: Symmetric Expiration Handling

As discussed in Section 3.5, some distributed software platforms have previously employed leases as a technique for resource management and memory management. In this context, lease grantors remain in control of the resource by maintaining the right to free the resource once the lease expires. Usually *only* lease grantors are aware of a lease expiration so that they can free resources created during the lease agreement. When employing leasing as a failure detection abstraction that allows MANET applications to distinguish intermittent from permanent failures, lease expiration represents the permanent failure of a service. Hence, *both* lease grantors and holders should be aware of a lease expiration in order to allow them to gracefully terminate their collaboration.

In our running example, the `session` object is clearly only relevant within the context of a single music library exchange. If the exchange cannot be completed (e.g., due to a persistent network partition), the session object and the resources allocated during the session (e.g., the partially uploaded library index) should be eventually reclaimed. From the lease holder perspective, once a lease expires, it should also be possible to perform some compensating actions to deal with the permanent failure. From the lease holder perspective, when the lease on the session in device `B` expires, the application running on `A` notifies the user of the failure of the exchange. If the failure was wrongly approximated, and the peer is encountered again in the future, a new session will be established. However, other compensating actions are also possible. For example, the application could decide to keep the partially received music library instead and resume the session if the peer is encountered again in the future.

To sum up, a leasing model for MANETs should allow both lease holder and lease grantor to react to the termination of their logical connection so that they can perform application-dependent failure handling.

## 6.2 Leasing and Programming Languages: A Proposal

To meet all these aforementioned criteria, we now introduce the design and implementation of a leasing model for MANETs. The essence of our leasing model is the combination of an ambient-oriented style of communication with a lease programming language concept that enables developers to express different leasing patterns among distributed processes. We will henceforth refer to this leasing model as *ambient-oriented leasing*. In order to be as general as possible with respect to the kinds of distributed interactions, we integrate the lease concept reflectively, i.e., as a first-class object that exposes a number of methods that can be overridden, enabling developers to extend the set of built-in leasing patterns. We first introduce some terminology to precisely define the key properties of our lease concept.

**Definition 1 (Lease)** A lease is a contract between a lease grantor and lease holder that allows the lease holder to use a *resource* for a specific period.

A lease should be regarded as an coordination abstraction that represents an explicit agreement about the access to the service offered by a owner of a resource to a resource user for a specific period. The lease grantor promises to offer a service to the network in exchange for the lease holder's promise to use the service object complying with the lease agreement. The lease grantor can restrict the access to the leased object when the lease agreement is violated. Representing a lease as an enforceable contract is of great significance in a mobile setting in which the interest of the communicating parties may diverge, and they cannot rely on a centralized authority to mediate their interactions.

In our context, a resource is an object which provides some type of functionality or service (which we refer to as a service object). A lease specifies an agreement concerning the right to access and execute some actions on a service object, e.g., to modify it, or perform method invocations on it. However, a lease does not provide ownership rights to the lease holder. In general, either the lease grantor is the owner of the object being leased and retains its ownership, or the lease grantor just manages the resource and does not have ownership rights itself.

In Section 6.1.2 we argued that there is no single right leasing semantics for *all* kinds of collaborations in a MANET application. This observation has lead us to scrutinize the different aspects of leasing. Next to identifying the service object being leased, a lease consists of two essential components along which the behaviour of a lease may vary: a *term* for which the lease holder can access the resource, and detailed *statements* that rule the lease agreement. The result of composing the behaviour of the lease term and statements gives rise to different leasing variants. Figure 6.2 gives a graphical overview of all the components by means of a feature diagram. In the remainder of this section, we define each of these components and present the most important properties identified for each component.

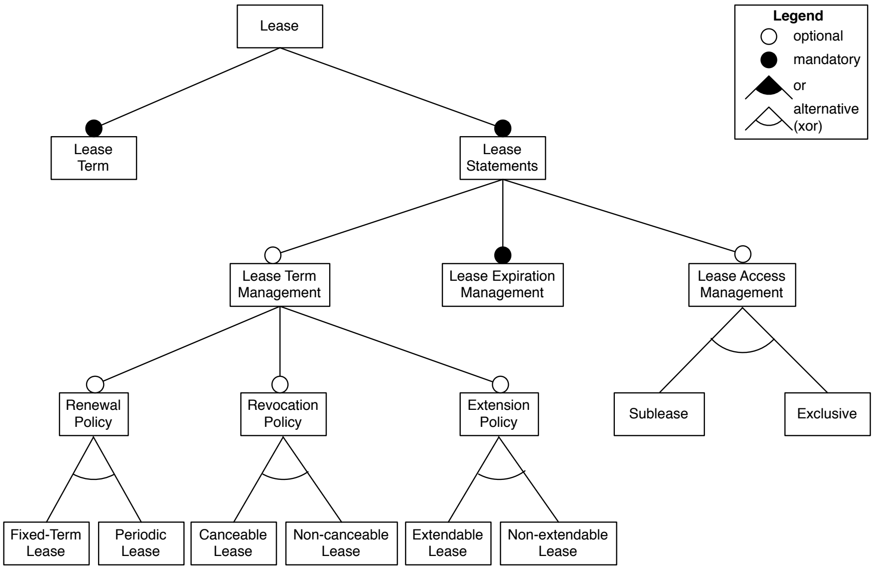


Figure 6.2: Feature diagram representing the lease concept.

**Definition 2 (Lease Term)** The lease term (also called *lease period*) denotes a specific time duration for which a lease agreement is in force.

The actual length of the lease term can be defined either by means of a definite time interval or by means of a characteristic function that determines the duration based on the occurrence of some event. In this work, a term lasts until the specified time interval elapses (e.g., 10 minutes, one hour, etc). This is also known in as a *time-based term*. The study of *conditional* terms is outside the scope of this dissertation (that we discuss in Section 6.9).

**Definition 3 (Lease Statements)** The lease statements detail the *terms and conditions* of the lease agreement specifying the nature and scope of a lease.

By nature, a lease entails a two-party agreement in which the lease holder has the right to access the resource for the lease term, and the lease grantor retains ownership of the resource. However, specific rights can be given to lease holder and third parties (objects which do not take part in the original lease agreement). The lease statements describe the set of properties that can be retained on the lease agreement. By default, the lease statements include a lease expiration management component to enable both lease holder and grantor to be notified of the expiration of a lease. This component is essential to enable developers to act upon a failure and perform the necessary corrective actions. In addition, the lease statements consist of the following two components:

- the *lease term management* which indicates the set of operations that a lease offers to manage the validity of the lease agreement.
- the *lease access management* which indicates the set of operations that a lease offers to a lease holder to create and transfer leases to third party objects.

We now further detail the behaviour of each of these components in the following subsections.

### 6.2.1 Lease Term Management

Lease term management consists of a number of policies that determine the ability to prolong or cease the validity of the lease term. Typically, two operations can be offered when a lease is active: a lease may be renewed (i.e., its term gets prolonged) or revoked (i.e., its term gets ended). When a lease expires, lease holders can be offered the ability of extending its term. As such, lease term management consists of three different sorts of policies with respect to lease renewal, revocation and extension. We further describe each policy in what follows.

**Lease Renewal Policy** determines the type of lease agreement with respect to the ability of a lease holder to extend the validity of the lease term before it has expired. There are the following choices regarding the renewal policies:

- **No renewal** A lease with a no renewal policy, called a *fixed-term lease*, implies that the lease expires automatically at the end of the specified term.
- **Renewal** A lease with a renewal policy, called a *periodic lease*, implies that the lease can be renewed for a certain duration before its current term elapses. To this end, the lease provides a method to renew the lease at the lease holder's discretion.

A lease holder can renew a periodic lease either at the same rate, in which case the current lease term is used to prolong the lease, or at a different rate, in which case the renewal is requested with a new lease term. The latter may require a renegotiation of the lease term with the lease grantor if a *longer* term is provided in the renewal request. For instance, it is not possible that a periodic lease with a term of 5 minutes is renewed with a 10-minute term since the original lease term agreed is shorter than the requested segment length.

A periodic lease can be created with a *renewal decision rule* which determines under which circumstances a renewal request can be granted. For example, a renewal decision rule may be set to limit to the number of times a lease holder can request a renewal. If the application already exceeded the limit, the renewal request is denied. In this case, no communication is required between the lease holder and grantor. However, other decision rules may be based on information only known by the lease grantor requiring communication, e.g., if the renewal condition is based on the current number of lease holders for the resource in the system. Naturally, since a fixed-term lease cannot be renewed, it does not require a renewal decision rule.

**Lease Revocation Policy** determines the lease agreement with respect to the ability for a lease holder to put an end to the validity of the lease term before it has expired. There are the following choices regarding the revocation policies:

- **No revocation** A lease with a no revocation policy, called a *non-cancelable lease*, implies that the lease cannot be revoked.
- **Revocation** A lease with renewal, called a *cancelable lease*, implies that the lease can be revoked before its current term elapses. To this end, the lease provides a method to revoke the lease at the lease holder's discretion.

Similarly to periodic leases, a cancelable lease can be created with a *revocation decision rule* which determines under which circumstances a revocation request can be granted. However, it is not usual that a cancelable lease limits revocation operations using a revocation decision rule because revoking a lease allows for opportunistically releasing resources.

**Lease Extension Policy** determines the type of lease agreement with respect to the ability for a lease holder to extend the validity of the lease term when it has expired. As argued in the previous section, a lease can only estimate permanent failures. Since in some cases leases may falsely treat the failure of a slow process as a permanent one, it is important to consider means to extend a lease rather than immediately invalidating it. This allows us to minimize the cost of requesting new leases if the original lease term provided a wrong estimation of a failure. There are the following choices regarding the extension policies:

- **No extension** A lease with a no extension policy implies that the lease cannot be further extended once it expires. The lease grantor needs to ask a new lease if he requires access after the expiration of the lease.
- **Extension** A lease with extension, called a *extendable lease*, involves a termination protocol (similar to TCP connection shutdown) upon lease expiration in which the lease holder may renew the lease term.

Naturally, a lease extension policy is not required for fixed-term leases since they cannot be extended.

### 6.2.2 Lease Access Management

Recall that a lease gives the right to access an object during the lease term, i.e., the lease holder has *permission* to access the object. Lease access management specifies the right a lease holder has to delegate a lease (and thus the right to access the object) to a third party. There are the following choices regarding lease access management:

**Sublease** A sublease implies that the lease grantor has the right to create a new lease from the initial lease, and transfer it to a third party object. This means that the lease holder is leasing a service object, but also subleasing it simultaneously. The new lease is ruled by the same lease term and statements as the initial lease, i.e., a lease holder cannot grant more rights than the acquired ones by the initial lease agreement. For example, if the lease holder has a time-based lease that lasts, say, 5 minutes, and he shares its lease once there is only 2 minutes left, the third party object only obtains a lease for the remaining duration of the term (i.e., 2 minutes).

**Exclusive** An exclusive lease implies that the lease grantor cannot sublease the lease to a third party. This means the lease explicitly denotes a exclusive two-party agreement between the lease grantor and holder which cannot be shared.

In MANETs, sharing leases allows cooperation without requiring any fixed infrastructure. However, when a lease holder shares a lease with a third party object, the object may gain the *authority* to access an object, and affect it. In the literature, the principle of least authority (POLA) recommends granting only the authority that an object needs to carry out its job [MS03]. This discipline helps coordination of processes by reducing the number of cases that objects need to protect themselves against each other to remain consistent [Mil06]. Making explicit in the design of a lease which kind of access third parties can gain to a leased service sets the basis for building abstractions practicing POLA for dealing with malicious behaviour that may lead to arbitrary failures. However, their design and implementation are outside the scope of this thesis.

### 6.2.3 Summary

In this section, we have introduced the concept of a lease in an abstract way together with the necessary terminology describing the important components of a lease. Rather than designing a lease as one uniform abstraction, we have identified a number of components which form a lease. The main rationale for such a decomposition is that we employ leases as the heart of a failure handling model for MANETs fulfilling the requirements described in Section 2.4. The composition of lease term and lease statements constitutes the kind of lease variant offered to a resource which lease holders must agree to comply with in order to use that resource. Each component will be incorporated in our leasing model presented in Section 6.3. We conclude the description of our lease concept by discussing two components of a lease which we did not explore in this work as extension points, namely, lease expiration management, and exclusive access rights.

#### 6.2.3.1 Lease Expiration Management

In [JK00], Jain and Kircher describe a lease variant in which a lease can be created with no expiration, i.e., the lease holder must cancel the lease explicitly when it no longer needs the resource associated with the lease. This variant is not supported in this work because it would not allow objects to agree on some criteria to delimit their logical communication in the face of partial failures, which defeats one of the most beneficial properties of using leasing at the heart of a failure handling model.

In a legal context, a lease may have addenda or options to be exercised once a lease expires. For example, when applying leases to car purchases, the lease holder may gain ownership of the car once the lease term ends. In our lease concept, we do not incorporate addenda which allow a lease holder to acquire ownership rights to the resource. Although ownership is an interesting concept to explore as a mechanism to organize the distributed object reference graph, this option is out of the scope for this work. We further discuss this in the avenues of future research (cf. Chapter 12).

Our concept of lease does allow lease holders to be notified once the lease term expires. In particular, those lease holders who registered an interest in the expiration of a lease will be notified. Notifying lease holders is essential in a leasing model for MANETs in order to allow programmers to apply some corrective actions (as we previously argued in the symmetric expiration handling requirement in the previous section). While some leasing approaches provide these semantics as a lease variant [AKS05], we consider them to be part of the core concept of a lease. As shown in Figure 6.2, rather than making expiration lease management a component whose functionality may be withdrawn from a lease variant, our leases expose it as a mandatory component of the lease concept.

#### 6.2.3.2 Exclusive Access Rights

In the context of distributed file caching where leases were first introduced in software engineering, leases confer exclusive possession over a time period. As a result, there exists only *at most one* lease active per resource at a time. A client cannot acquire a lease for a lease term that overlaps with the term of a lease previously granted i.e., no two leases have overlapping lease terms.

In contrast, we admit that several distinct leases for the same resource may coexist in the system. The rationale behind this design decision is that a lease represents a promise of the service that an object offers to the network. Since the different parties taking part in a collaboration in a MANET may have different interests and properties (e.g., some users may be more privileged than others), allowing different leases for the same resource allows us to provide several lease agreements for a service depending on the communicating parties. Although in this work we do not explore leases that convey exclusive access, such semantics could be refectively encoded on top of the basic lease concept.

## 6.3 Leased Object References

In this section, we introduce the key concepts of our leasing model which integrates the lease concept described in the previous section into a distributed object-oriented model, leading to the concept of *leased object references* (or just leased references). In a nutshell, a leased reference is a remote object reference that limits the communication between client and service objects based on a lease agreement. Alesky et Korthaus remarked in [AK06] that a lease acts as a kind of “connecting link” between the lease holder and resource. In a distributed object-oriented model, a remote object reference acts as a communication link that carries messages from a client to a service object they refer to. As such, a remote object reference can naturally serve as a lease in which the client object plays the role of the lease holder, and the service object plays the role of the resource. A leased reference is thus a remote object reference that limits the communication between client and service objects based on a lease agreement, i.e., it carries messages for a limited duration.

When a client first references a service object, a leased reference is created and associated to the service object. From that moment on, the client accesses the service object *transparently* via the leased reference until it is no longer valid. Each side of the leased reference has a lease object initialized with a term in order to keep track of the validity of the lease agreement. When the lease term has elapsed, the leased reference is said to *expire* and the access to the service object is terminated. The lifetime of the lease can be explicitly controlled by renewing or revoking the leased reference before it expires. Once a leased reference expires, *both* the client and service object know that access to the service object is terminated. Leased references define dedicated object serialization semantics to ensure that third party objects accesses to the service object are also leased.

### 6.3.1 Time-decoupled Object References

In order to abstract over the transient disconnections inherent to MANETs, a leased reference decouples the client object and the service object in time. This means that a client object can send a message to the service object even if the leased reference is disconnected at that time. Similar to a far reference, client objects can only send messages to service objects *asynchronously*: when a client object sends a message to the service object, the message is transparently buffered within the leased reference and the client does not block. In essence, a leased reference decouples client and service objects in time, but only up until the period of time described by the lease term.

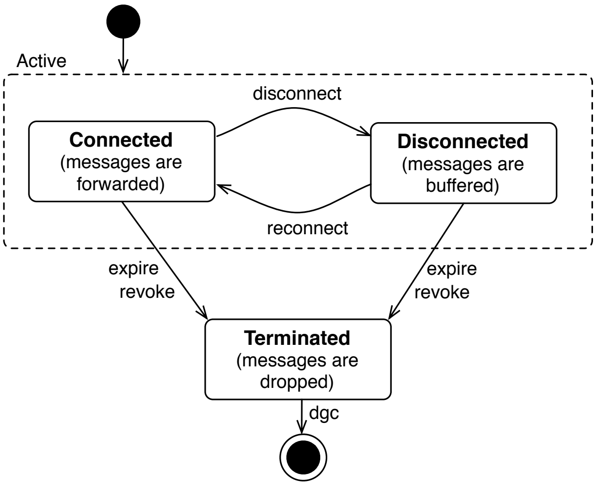


Figure 6.3: States of a leased object reference.

Figure 6.3 shows a UML-state diagram of the different states of a leased reference. In general, a leased reference is said to be *active* when its lease term has not elapsed yet, and terminated otherwise. While a leased reference is active, client objects can send messages to the service object which are buffered in the leased reference. When those messages are actually transmitted to the service objects depends on the underlying network connection in a similar way to Rover (cf. Section 3.3.1.2). When the leased reference is connected and active (i.e., there is network connection and the lease has not yet expired), it forwards the buffered messages to the service object. While disconnected, messages are accumulated in order to be transmitted when the reference becomes reconnected at a later point in time.

A leased reference becomes terminated either when the lease term expires, or when the lease term is explicitly cancelled by means of a revoke operation. Only once a leased reference terminates, the distributed garbage collector can cleanup the reference, and the service object may become subject to garbage collection. When the leased reference becomes terminated, the client loses the means of accessing the service object via the leased reference. Any attempt in using it will not result in a message transmission since the leased reference behaves as a *permanently* disconnected remote reference.

### 6.3.2 Leased Reference Kinds

Rather than designing one kind of leased reference, we have designed a family of leased reference kinds, the behaviour of which varies according to how they manipulate the different components of the lease concept described in Section 6.2. In this section, we propose two default leasing variants provided by leased references. We defer the explanation of how new leased reference kinds can be composed until Section 6.7.

Leased references incorporate two leased reference variants that transparently adapt the lease term:

- A *renew-on-call* leased reference automatically renews the lease upon each message received by the leased object. This pattern has been inspired by the `renewOnCall` property of the .NET Remoting framework [Low03]. As long as the client uses the service object, the leased reference is transparently renewed by the system.

- A *revoke-on-call* leased reference automatically revokes the lease upon processing a message on the leased object. Such leased references are useful for objects adhering to a *single call* pattern, such as callback objects (such as futures). As previously explained, callback objects are often used in asynchronous message passing schemes in order for service objects to be able to return values. These callback objects are typically remotely accessed only once by service objects with the computed return value.

These variants are motivated by the trade-off that needs to be considered when choosing a lease duration between the amount of space maintained at the lease grantor side, and the messages that need to be sent by the client for managing the lease term. The shorter the lease term is, the lesser space the system needs to allocate less space. However, client objects need to send more messages to maintain their lease. In contrast, the longer the lease duration is, the system needs to allocate more space, but less messages need to be sent for lease term management. While a renew-on-call leased reference reduces the overhead of lease term management by granting longer leases to clients that have prolonged interest in the service object, a revoke-on-call variant reduces the amount of space the system needs to maintain by opportunistically releasing resources.

### 6.3.3 Symmetric Expiration Handling

In order to support symmetric expiration handling, we allow observers to be registered on leased references that trigger upon the lease expiration. In contrast to existing leasing models, those observers can be registered *at both sides* of a leased reference. This allows client and service objects to treat a failure as permanent (i.e., to detect when the reference is permanently broken) and to perform appropriate compensating actions. At the service side, this has important benefits for memory management. Once all leased references to a service object have expired, the object becomes subject to garbage collection once it is no longer locally referenced.

Because both sides of a leased reference have a timer, no communication with the server is required in order for a client to detect the expiration of a leased reference. However, having a client-side and server-side timer introduces issues of clock synchronisation. Keeping clocks synchronised is a well known problem in distributed systems [TS01]. This issue is somewhat more manageable with leases since they use time intervals rather than absolute time and the degree of precision is expected to be of the magnitude of seconds, minutes or hours. Once the leased reference is established, the server side of the reference periodically sends the current remaining time by piggybacking it onto application-level messages. At worst, the asynchrony causes a leased reference to be temporarily in two inconsistent states: either the client-side of the reference expires while the server-side is still active, or the client-side of the reference is active while the server-side expired. In the first case, a client will not attempt a lease renewal and thus, the server-side timer will eventually expire as well. In the second case, when a client requests a lease renewal, the server will ignore it and the client-side timer will expire soon thereafter. When the server-side timer is expired, the client perceives the remote object as disconnected.

## 6.4 Leased Object References in AmbientTalk

We now describe a concrete instantiation of leased object references in AmbientTalk by means of the mobile music player application introduced in Section 6.1.1. Leased references have been conceived as a set of AmbientTalk language constructs built on top of the transmitter-receptor model explained in Chapter 5. Table 6.1 summarizes all the language constructs. In the course of the following sections, we use this table as a reference to explain our language constructs. In Section 6.7 we will describe how programers can create custom leased reference kinds and added them to the language.

### 6.4.1 Declaring Leased References

We have integrated leased references in AmbientTalk as a custom referencing abstraction which is ruled by a certain lease agreement. Our language support features constructs for creating leased references which correspond to basic leased references and the two variants described in the previous section. In this section, we describe the most basic form of a leased reference, a reference with a time-based term can be renewed and cancelled, in the context of our running example. Recall that the `session` object that represents the exchange process between two music players should be subject to leasing in order for both music players to gracefully terminate the exchange process in the presence of network failures. Such a `session` object can be leased as follows:

| | |
|---|---|
| *Built-in leased references* | |
| **lease:** interval **for:** object | creates a leased reference to a service object for the given time interval. |
| **renewOnCallLease:** interval **renewedWith:** intervalRenewal **withRenewalConditions:** closure **for:** object | creates a leased reference that is automatically renewed on every message sent to the given service object. The **renewedWith:** specifies the renewal time, and **withRenewalConditions:** takes a unary closure encoding whether a given message send renews the reference. |
| **revokeOnCallLease:** interval **withRevocationConditions:** closure **for:** object | creates a leased reference that is automatically revoked after the object receives a message. The **withRevocationConditions:** closure specifies whether a given message send revokes the reference. |
| *Lease term management* | |
| **renew:** leasedref | requests a renewal of the given leased reference. |
| **revoke:** leasedref | requests a revocation of the given reference. |
| **leaseTimeLeft:** leasedref | returns the lease time left for a reference. |
| *Failure handling listeners* | |
| **when:** leasedref **expired:** closure | installs a listener on the given reference triggered when it expires. |
| **whenever:** reference **disconnected:** closure | installs a listener on the given reference triggered whenever it disconnects. |
| **whenever:** reference **reconnected:** closure | installs a listener on the given reference triggered whenever it reconnects. |
| *Leasing strategies* | |
| **withLeaseStrategy:** strategy **do:** closure | activates a leasing strategy within the dynamic scope of its closure argument. |
| **withoutLeaseStrategy:** closure | deactivates a leasing strategy within the dynamic scope of its closure argument. |
| **lease:** interval | creates a receptor representing a *pass-by-lease* referencing strategy. |
| **renewOnCallLease:** interval **renewedWith:** interval | creates a receptor representing a *pass-by-renewOnCallLease* referencing strategy. |
| **revokeOnCallLease:** interval **withRevocationConditions:** clo | creates a receptor representing a *pass-by-revokeOnCallLease* referencing strategy. |
| *Due-type message passing* | |
| @Due(interval) | specifies a time-based delivery guarantee on a future-type message send. |
| **when:** future **becomes:** closure **catch:** TimeoutException **using:** e | registers two closures, closure and e, on a future which are applied when the future is resolved, or ruined with a TimeoutException. |
| *Leased message protocols* | |
| makeTimeBasedProtocol(interval) | returns an object encoding a time-based leased message protocol. |
| **withLeaseProtocol:** protocol **do:** closure | activates the given leased message protocol within the dynamic scope of its closure argument. |
| **when:** protocol failedWith: closure | installs a listener on the given protocol which is triggered whenever a message fails to meet its expected delivery guarantees. |

Table 6.1: AmbientTalk's extended programming API for ambient-oriented leasing.

```
def leasedRef := lease: 10.minutes for: session;
```

The **`lease:for:`** construct takes two arguments: the time interval (in milliseconds) corresponding to the duration of a time-based term, and an object corresponding to the service object to which the leased reference grants access. It returns a leased reference which is active for the given time period unless a renewal or revocation is explicitly issued. Note that the **`lease:for:`** construct is executed at the lease grantor side before the `session` object is exported to a client object. The construct actually returns the *receptor* object of a leased reference. Recall from Section 5.2.1, that a receptor is an object representing the end of a reference which is located at the service object's actor. When the virtual machine hosting the service objects hands out the `leasedRef` receptor to a client object, the client object receives a leased reference data type representing the *transmitter* object of a leased reference. In our running example, a music player application asks a remote peer to start a session to exchange its library index by sending it an `openSession` message which returns a new session object. The music player can then send song information to the remote peer via the obtained leased reference as follows:

```
session<-uploadSong("Mika", "Relax", ...);
```

Because leased references are actually remote object references, the client can use the leased reference as if it were the service object itself. The use of leasing is thus made transparent to client objects.

It is important to remark that the term of a leased reference starts ticking when the reference is handed out to the client object. After the term expires, access to the session is terminated and the leased reference expires. As shown in Table 6.1 we provide support for explicit manipulation of the lifetime of a leased reference. The **`renew:`** construct requests a prolongation of the specified leased reference with a new interval of time which can be different than the initial time while the **`revoke:`** construct cancels the given leased reference. Cancelling a lease is in a sense analogous to a natural expiration of the lease, but it may require communication between the client and server side of the leased reference. Note that such constructs can be issued at both ends of the leased reference, i.e., either on the receptor at the service object side (stored in `leasedRef`), or the leased reference data type at the client side (stored in `session`).

### 6.4.2 Language Constructs for Leased Reference Variants

In this section, we describe the language constructs provided to create leased references for the two leasing variants described in Section 6.3.2.

**Renew-on-call Leased References.** The **`renewOnCallLease:for:`** construct creates a leased reference which is automatically renewed on every asynchronous message sent to the service object. When no renewal is performed due to a network partition or in the absence of utilization, the default leased reference expires once its lease term elapses. In the running example, once a music player establishes a session with another music player to exchange their music library index, the session should remain active as long as the exchange is active, i.e., as long as `uploadSong` messages are received. A renew-on-call lease can be used to model that kind of collaboration for the `session` object. Figure 6.1 shows the complete code for opening a music sharing session employing a renew-on-call lease rather than a basic leased reference.

Listing 6.1: Opening a music sharing session.

```
1  def openSession(sessionCallback){
2    def senderLib := Set.new();
3    def session := object: {
4      def uploadSong(artist, title, ackCallback) {
5        senderLib.add(Song.new(artist, title));
6        ackCallback<-ok();
7      };
8      def endExchange(){
9        revoke: session;
10       def matchRatio := calculateMatchRatio(senderLib);
11       if: (matchRatio >= THRESHOLD) then: {
12         // notify user of match
13       };
14     };
15   };
16   def leasedRef := renewOnCallLease: 10.minutes for: session;
17   sessionCallback<-receive(leasedRef);
18 };
```

As previously mentioned, the `openSession` message is sent by a music player to a remote peer which returns a session object that can be used to start a library exchange. A session implements the `uploadSong` method to send song information and the `endExchange` method to signal the end of the library exchange. The session object is exported using a lease for 10 minutes which is automatically renewed each time it receives a message. The leased reference to the `session` object is revoked either implicitly if the lease time has elapsed or explicitly upon receiving the `endExchange` message indicating the end of the library exchange (line 9). Since the session object was only referred to by the leased reference, it can be reclaimed once the lease has expired. Any resources it transitively occupied such as the partially uploaded library of songs (i.e. `senderLib`) can be reclaimed as well.

Note that in our running example, the renewal time applied on every call is the initial interval of time specified as the first parameter in the **`renewOnCallLease :for:`** construct. We also provide an extended form of this construct (called **`renewOnCallLease:renewedWith:for:`**) which allows developers to specify a renewal time different than the initial interval. In its most complete form (shown in Table 6.1), the **`renewOnCallLease:for:`** construct also takes as argument a unary closure encoding a boolean predicate that needs to be satisfied in order for a message to renew the lease. Such extension is catered for applications requiring higher degrees of context-awareness to support renewal of leases based on contextual information. For example, the mobile music player application could renew the lease on the session object only if the device battery level is above a certain acceptable level. This could be expressed in AmbientTalk as follows:

```
def leasedRef := renewOnCallLease: 10.minutes
    withRenewalConditions: {|msg| batteryLevel()>BATTERY_THRESHOLD} for: session;
```

The `batteryLevel` is a helper function that retrieves the device battery level from the underlying runtime platform[1]. It is important to notice that the renewal conditions are only executed at the lease grantor side. Whenever the receptor of a leased reference receives a message from a client object, the closure is triggered (i.e., asynchronously applied to) taking the message object as argument. If the conditions hold (i.e., the closure returns `true`), the leased reference is conceptually renewed. In this example, the leased reference is renewed independently of the message being received as long as the battery level does not pass the minimum threshold.

[1] The battery level of Android devices can be monitored by registering an observer to the `ACTION_BATTERY_CHANGED` intent.

**Revoke-on-call Leased References.** The **revokeOnCallLease:for:** construct allows developers to create leased references that expire after the service object receives a single message. By default, the leased reference remains valid for only one asynchronous message send. However, if no message has been received within the specified time interval, the leased reference expires. As shown in Listing 6.1, the sessionCallback is passed in the openSession message to asynchronously receive a session object. A revoke-on-call lease can be used for unexporting this callback object upon receipt of the receive messages:

```
remotePlayer<-openSession(
  revokeOnCallLease: 10.minutes for: ( object: {
    def receive(session){
     /* start to exchange of its library via the session (explained later) */
    }
}));
```

A lease time of 10 minutes is specified to wait for the reply of the openSession message. If a disconnection would occur after the message was sent but before the receive reply was received, the session object could have already been allocated. Since a session's lease only lasts 10 minutes by default, it does not make sense to wait any longer for the reply. If the session callback's lease expires, the music library exchange terminates before it was actually started, requiring no additional cleanup code.

The default semantics of a revoke-on-call lease provided is catered for single-call objects such as the callback object in our running example. In its most complete form (shown in Table 6.1), the **revokeOnCallLease:for:** construct also takes as parameter a unary closure defining whether a given message send revokes the leased reference or not. It works analogously to the previously explained renew-on-call lease counterpart, but the net effect is to cancel the lease term rather than renewing it. Note that when employing revocations conditions, the leased reference may have processed more than one message before being cancelled. We will describe later in Section 6.5.1 how we have employed such support to integrate leasing into distributed computing to provide time-based message delivery guarantees.

### 6.4.3 Expiration Handling

As previously mentioned, leased references make leasing transparent to client objects which can access directly a service object by means of asynchronous message passing. However, it should still be possible to detect and react when a leased reference expires, as the lease expiration denotes the failure of the logical communication with a service object. To this end, we introduce a failure event handler which can be registered with a leased reference to be notify upon its expiration. The code excerpt below shows how a music player can detect that a session with a remote music player expires.

```
when: session expired: {
  system.println("session timed out.");
  // clean the partially received music library
}
```

The **when:expired:** construct takes as arguments a leased reference and a nullary closure that is asynchronously triggered when the underlying implementation detects that the leased reference has expired. In the example, the failure handler is installed at the server side so that if the exchange cannot be completed the resources a session transitively keeps alive can be cleared. However, **when:expired:** handlers can also be placed at the client side in order to allow both client and service objects to perform failure handling upon leased reference expiration. Note that by default the term of a leased reference is not extendable, as such, the **`when:expired:`** handlers are only triggered once per each leased reference.

As shown in Table 6.1, we have adapted AmbientTalk/2's default failure handlers to work with leased references. Client objects can install **`whenever:disconnected`** and **`whenever:reconnected`** handlers on leased references to be notified whenever a reference becomes disconnected or reconnected, respectively.

### 6.4.4 Leasing Strategies

Integrating leasing into the remote object reference abstraction aligns well with a distributed object-oriented model of computation since remote references are the unit of object designation. However, this implies that developers must express leasing semantics in a per-object basis, hindering the reuse of leasing semantics among groups of objects. To cater for this case, we introduce the notion of *leasing strategies* that allows to apply the same leasing semantics to all objects exported within a dynamic execution extent. Before describing leasing strategies, we introduce a concrete example in the context of our running example that we use to explain our language support.

#### 6.4.4.1 Motivating Example: Listening Nearby Music Libraries

Consider an extension to the mobile music player application, in which users can also *play* songs from remote music libraries of nearby peers. To this end, they can open a music playing session with remote peers, which subsequently sends their songs so that the user can play songs locally on his mobile device. In this case, the song objects shared during the music player library exchange include an audio track next to the metadata associated with a song (the title, and the artist as defined in Listing 5.5). In order to optimize the mobile device's resource, an audio track is only copied to the remote device if the end-user wishes to play the song (since they are potentially large objects). To this end, they are passed by *lazy reference* as explained in Section 5.2.4 (see Listing 5.6 for the audio track object definition). This means that the peer first receives a remote reference to the audio track. The audio track is then fetched and copied to the remote device if necessary. Audio tracks should only be accessible as long as the session is active. As such, each remote reference to audio tracks created during the session should be also leased.

#### 6.4.4.2 Language Support for Leasing Strategies

Leasing strategies give control over the propagation of the leased references that are created within the dynamic extent of a body expression. They have been inspired by the notion of layers of the context-oriented programming ContextL [CH05]. In ContextL, layers group related context-dependent behavioural variations. In our work, a leasing strategy represents the kind of leased reference by which an object is remotely designated (see Section 6.9 for a detailed discussion of both abstractions). A leasing strategy is automatically applied when an object is exported to a remote object by passing it as parameter or return value in an asynchronous message send. Similar to ContextL layers, we provide constructs to activate and deactivate leasing strategies dynamically at runtime. As shown in Table 6.1, the **`withLeaseStrategy:do:`**

Listing 6.2: Opening a music listening session.

```
def openMusicListeningSession(sessionCallback){
  withLeaseStrategy: (renewOnCallLease: 60.minutes) do: {
    def musicListeningSession := object: {
    def index := 1;
      def fetchNextSong(songCallback){
        if: (index < myLib.length) then: {
          def song := myLib.get(index);
          songCallback<-receiveSong(song);
          index := index + 1;
        }
      };
    };
    sessionCallback<-receiveSession(musicListeningSession);
  }
}
```

and **`withoutLeaseStrategy:`** constructs allow for the activation and deactivation of lease strategies within the dynamic scope of its closure argument.

Listing 6.2 illustrates the usage of **`withLeaseStrategy:do:`** in the running example. The protocol for sharing songs within a music listening session works a bit different than the music library exchange protocol depicted in Figure 6.1. In this case, when a remote peer receives session, it receives a leased reference to the `musicListeningSession` object which can use to start fetching songs objects from the remote peer by sending `fetchNextSong` messages. The **`withLeaseStrategy:do:`** construct is used to apply a renew-on-call leasing strategy to the `musicListeningSession` object, and all audiotrack objects passed along with the `song` objects in the `receiveSong` asynchronous message. The first argument defines which leasing strategy to apply, and the second argument is a block closure in which the given leasing strategy is deployed. A leasing strategy represents a *pass-by-lease* referencing abstraction which should be installed to each object passed to remote objects within the dynamic extent of the **`withLeaseStrategy:do:`** construct. More concretely, a leasing strategy corresponds to a receptor of a leased reference which has not been installed to a service object yet. When an object is passed among within the dynamic extent of the **`do:`** closure, the system creates a new instance of the given receptor prototype and binds it to the object being passed, so that the object is passed by the corresponding leased reference. As shown in Table 6.1, we provide constructs that return a leasing strategy for the three built-in leased reference variants. In this example, **`renewOnCallLease:`** is a variation on the **`renewOnCallLease:for:`** construct returns a first-class renew-on-call receptor. It is possible to define custom leasing strategies. This is discussed later in Section 6.7.

As mentioned, the activation of a leasing strategy affects the behaviour of the program within the dynamic extent of the **`do:`** closure. We now detail the scoping semantics associated to the activation of the leasing strategy. The given leasing strategy is propagated to:

(1) any object passed within the control flow of the **`do:`** closure,

(2) any object created in the **`do:`** closure (even if they are passed to a remote object after executing the closure),

(3) any object passed within the execution of an asynchronous message processed by an object created in the **`do:`** closure.

In our example, the renew-on-call leasing strategy is applied to the musicListeningSession object according to (1) (because it is passed in the receiveSession message (line13)), and to the audiotrack objects passed along with the song objects sent in receiveSong (line 8) according to (3) (because they are passed within the code executed for the receiveSong message sent to an object passed with the given leasing strategy (musicListeningSession). When the control flow returns from the dynamic extent of the **do:** closure, the leasing strategy is deactivated, and objects are exported according to the default parameter semantics (which as usual can be overridden in a per-object basis by means of the *__lease:for:__ constructs introduced in the previous section).

It is possible to nest **withLeaseStrategy:do:** construct which means that the most inner leasing strategy is applied within the dynamic scope of the **do:** closure, and once outside this inner layer, the outer leasing strategy will be applied again. In other words, nesting the **withLeaseStrategy:do:** constructs causes a switch of the leasing strategy applied during the execution of the **do:** closure. Within the dynamic context of a **withLeaseStrategy:do:** construct, the leasing strategy can be deactivated by means of the **withoutLeaseStrategy:** construct. It takes only one argument; a block closure in which the current active leasing strategy is deactivated. This has the effect of restoring the outer leasing strategy (if any defined) or the top (default) leasing strategy to objects exported within the scope of the given closure. While the **withLeaseStrategy:do:** construct delimits the scope boundaries of the deployment of a leasing strategy, the **withoutLeaseStrategy:** construct support its local *undeployment*, i.e., it hides the export of certain objects within those boundaries.

## 6.5 Integrating Leasing with Future-type Message Passing

Limiting the lifetime of remote references using leased references allows developers to make assumptions about the timing behaviour of the application with respect to distributed communication, i.e., the time period that a remote object promises to be accessible during an interaction. Sometimes developers need to make assumptions about the timing behaviour of the application in respect to distributed *computation*, the time period that it is acceptable for processing a message sent to a remote object. Note that processing a message involves the delivery of a message to a remote object, its computation at the remote side, and the sending of a message with the return value of the computation. In this section, we describe how we integrate the lease concept into message passing in order to allow developers to specify *time-based delivery guarantees* on individual messages, i.e., how long to await the return value of a message.

### 6.5.1 Due-type Messages

In the previous section, we used an explicit *callback* object (also known as the *customer* of the message being processed [Agh90]) with a method to obtain the result of the openSession asynchronous message. This is motivated by the fact that in AmbientTalk an asynchronous message send has no return value by default. To avoid forcing programmers to rely on explicit, separate callback objects, future-type message passing was introduced in AmbientTalk (as we described in Section 4.3.1). We can rewrite the asynchronous invocation of openSession using futures as follows:

```
def sessionFuture := remotePlayer<-openSession()@FutureMessage;
when: sessionFuture becomes: { |session|
  // open session with remote the player (explained later)
}
```

We have integrated leasing into futures by introducing a new kind of message annotation called `@Due` which puts an upper bound on the resolution of the future associated to the message. As such, a *due-type message send* denotes a "bounded" future-type message send whose limit is denoted by a time-based lease. To be more precise, `@Due` takes as parameter the time interval denoting the expected deadline within the future attached to the message is expected to be resolved. We can use the `@Due` annotation in the above example to denote time the application is willing to wait for a music exchange session to be started as follows:

```
def sessionFuture := remotePlayer<-openSession()@Due(10.minutes);
when: sessionFuture becomes: { |session|
  // open session with the remote player (explained later)
} catch: TimeoutException using: { |e|
  notification("unable to open a session.");
}
```

The `@Due` annotation makes the `openSession` message send immediately return a future (stored in the variable `sessionFuture`) which is *passed by leased reference* to the receiver object (`remotePlayer`), rather than occupying the default *pass-by-far-reference* semantics. More concretely, the future is passed using a single-call leased reference initialized with a time-based term (of 10 minutes in this example). If the return value of the message being processed is received within the lease term, then the future is resolved with the return value. As explained in Section 4.3.1, when a future is resolved, the **`becomes:`** closure is applied to the return value. In this example, the `session` variable refers to a leased reference to the remote session object). If the return value of the message being processed is not received within the lease term, then the future is automatically ruined with a `TimeoutException` exception. As a result, the **`catch:`** closure is applied to the `TimeoutException` exception.

The leased reference by which a future is passed to the receiver object of an asynchronous message either terminates because the lease term expires, or upon the reception of the return value. When performing failure handling on the future-type message send it is important to notice two things. First, if the computation of the message failed (e.g., if `openSession` raises an exception), then the future will be ruined with the raised exception (rather than a `TimeoutException` exception) and the leased reference becomes terminated. Second, it may be possible that the message was successfully processed by the receiver of the message, but the return value arrived after the lease term of the reference expired. In this case, the return value is lost since the future has been previously ruined with a `TimeoutException` exception. Note that specifying a **`catch:`** block for the `TimeoutException` is equivalent to install a **`when:expired:`** observer on the future's (server-side) lease.

Listing 6.3 shows what happens when the future `sessionFuture` is properly resolved to a session object. Recall from the previous section that `session` refers to a leased reference to the session object. A music player then sends all of its own songs one by one to this remote session object via the `uploadSong` method as previously described in Figure 6.1. After all songs have been sent, the `endExchange` method is invoked to signal the end of the library exchange protocol.

The auxiliary function `sendNextSong` sends the music player's songs one by one to the remote `session` object. This serial behaviour is guaranteed because each sub-

Listing 6.3: Implementation of the music library exchange protocol.

```
def sessionFuture := remotePlayer<-openSession()@Due(10.minutes);
when: sessionFuture becomes: { |session|
  def iterator := myLib.iterator();
  def sendNextSong() {  // auxiliary function to send each song
    if: (iterator.hasNext()) then: {
      def song := iterator.next();
      def ackFut := session<-uploadSong(song.artist,
              song.title)@Due(leaseTimeLeft: session);
      when: ackFut becomes: { |ack|
        sendNextSong();
      }catch: TimeoutException using: { |e|
        notification("stopping exchange: " + e);
      };
    } else: {
      session<-endExchange();
    };
  };
  sendNextSong();
} catch: { |e|
  notification("unable to open a session due to " + e);
}
```

sequent `uploadSong` message is only sent after the previous one returned an acknowledgement. Since the return value of the `uploadSong` message is only useful in the context of the current library exchange session, it only makes sense to wait for the future resolution for the remaining duration of the session. Such duration can be extracted by means of the **`leaseTimeLeft:`** construct which takes as parameter the leased reference to the session object. If the future is not resolved within this time deadline, the library exchange is stopped without requiring additional cleanup code.

### 6.5.2 Leased Message Protocols

The integration of leasing into future-type message passing by means of the `@Due` annotation, illustrates that low-level memory management concerns (e.g., the callback objects) can be cleanly incorporated into more high-level abstractions. However, annotating each asynchronous message send with `@Due` annotations puts extra burden on developers. This can be alleviated by using leased references, as all messages sent via a leased reference are sent using the same timing assumptions (expressed at the lease reference declaration). Many times messages sharing the same timing assumptions are not sent to the same receiver, but they belong to a *semantic* protocol which involves the collaboration of a number of remote objects. Manually calculating those timing assumptions can quickly lead to intricate code as applications become more complex. To make leasing practical for distributed computation (expressed by messages), we introduce the notion of *leased message protocols* which abstract the timing dependencies of a *group* of due-type message sends. Before explaining leased message protocols, we detail the pattern they abstract by means of a concrete example.

#### 6.5.2.1 Motivating Example: Context-Aware Music Playlists

Consider an extension to our mobile music player application. Next to exchanging the music library, the extension provides a context-aware sharing functionality which combines the musical preferences of the nearby's friends in order to construct an acceptable playlist for all nearby users. A user first needs to inform the music player application

Listing 6.4: Implementation of the music genre voting protocol.

```
def broadcastVote(genre, maxVoteTime) {
  def [future,resolver] := makeFuture();
  def receivedVotes := Map.new();
  def retrieveVote(player, timeLeft) {
    when: player<-askToVote(genre)@Due(timeLeft) becomes: { |vote|
      receivedVotes.put(player, vote);
    };
  };
  nearbyFriendPlayers.each: { |player|
    retrieveVote(player, maxVoteTime);
  };
  def alreadySent := nearbyFriendPlayers.copy();
  def voteStartTime := now();
  def sub := whenever: Player discovered: { |player|
    if: !alreadySent.contains(player) then: {
      alreadySent.add(player);
      def timeLeft := maxVoteTime - (now() - voteStartTime);
      when: player<-getUsername()@Due(timeLeft) becomes: { |username|
        if: (isInFriendList(username)) then: {
          def timeLeft := maxVoteTime - (now() - voteStartTime);
          retrieveVote(player, timeLeft);
        }
      }
    }
  };
  when:  maxVoteTime elapsed:{
    sub.cancel();
    resolver.resolve(receivedVotes);
  };
  future
};
```

about his musical preferences (e.g., one or several music genres). When a user enables the *context-aware music* playlist, the application starts a poll among all nearby peers about a certain music genre. When a peer gets asked to vote, it submits its result to the originator of the poll. The originator waits a while before processing the received results so that all peers have the chance to vote.

#### 6.5.2.2 Manual Implementation

Listing 6.4 shows the relevant code required to implement the music genre voting protocol[2]. In order to implement a poll among all nearby music players, one needs to define data structures to keep track of which friends are connected in the proximity and who rated certain genres. Assume a vector `nearbyFriendPlayers` storing references to the *interface* object of another friend's music player application discovered in the network. Recall from previous section that the interface object had one method `openSession` to start the music library exchange protocol.

The `broadcastVote` function takes as parameter a textual description of the genre to vote, and a time interval `maxVoteTime` denoting how long the originator of the poll will wait for results. It returns a future whose value is a hashmap associating the friends who replied to the poll to their vote. The future is resolved when the `maxVoteTime` interval elapses. The application can then calculate the ratings of a particular genre depending on the current number of friends connected, and decide on a wining genre

[2] This code has been previously employed in other AmbientTalk publications, e.g., in the team-based mobile game scenario in Van Cutsem's dissertation [Van08]

or broadcast a new poll on a different genre.

More concretely, the `broadcastVote` function first asks the players which were already stored in the `nearbyFriendPlayers` vector at the time of starting the poll to vote on a given genre (lines 9-11). In addition, it installs a new discovery handler (line 14) so that the music player applications of friends which are discovered while the poll is active are also asked to vote (i.e., friends who become online after the poll was initiate and before the `maxVoteTime` elapses). Since retrieving the vote of a newly discovered friend is similar to retrieving the vote of a previously discovered one, the shared code is put in the `retrieveVote` function. When discovering a new music player in the neigbourhood, the username of the discovered peer is asked in order to check if it belongs to the friend lists of the user[3].

#### 6.5.2.3 The Problems

Listing 6.4 depicts in blue the code dedicated to deal with the timing issues of the voting protocol. Despite being a relatively small example, it illustrates two issues that the programmer has to face when explicitly encoding the timing assumptions of message sends in the application code.

- When implementing a distributed protocol like voting, the several messages sent during this interaction share the same timing assumptions. However, using only the `@Due` annotation forces programers to manually calculate the time left on each stage of the protocol, and percolate it through the dynamic extent of the application to make the corresponding asynchronous message sends see the correct updated value in its scope. In the voting example, the upper bound on the time allowed to vote for newly discovered players needs to be manually calculated as the later a new player is discovered, the fewer time to vote the player has. Calculating the time left to vote (stored in the `timeLeft` variable) is done by subtracting from the original `maxVoteTime` interval, the time elapsed between the start of the poll and the current time (retrieved by means of the `now` helper function). Such scattering of timing assumptions in the application code leads to intricate code. In addition, it obfuscates the dependencies that exists between the different messages sends forming part of the protocol.

- Second, developers also need to deal with the failure handling code when a part of the whole protocol fails to meet the timing expectations. In the voting example, the application proceeds even if some votes were not collected within the time boundaries, so no failure handling code is required when gathering the results. Other applications may need to restart part of the protocol if one of the intermediate stages fail. When the whole protocol finishes, the programmer may also to need to apply some cleanup or corrective actions. In this example, the application just executes some cleanup code to cancel the discovery handler for new music players, and proceeds with the gathered results. Depending on the application and the kind of protocol being implemented, different protocol management semantics must be implemented.

In short, the programmer must has to manually implement a layer of abstraction to encode the timing assumptions of distributed computation and how to react when those

[3] We assume that the user friend list is retrieved from the device's contact application, e.g., by means of the `android.provider.Contacts` class in the Android platform.

assumptions are not met. The above code clearly represents a pattern that programmers will have to deal with when encoding applications protocols involving several communicating parties. However, this pattern cannot be easily modularized because the concrete upper bound to apply to an asynchronous message may depend on the computational context of the message send. In the following section, we introduce the leased message protocol abstraction, the goal of which is to encapsulate the timing assumptions on message sends into a separate abstraction.

#### 6.5.2.4 Language Support for Leased Message Protocols

A *leased message protocol* gives control over the asynchronous messages sent within the dynamic extent of a body expression. We represent a leased message protocol as an object that automatically traces asynchronous message sends within a dynamic execution, and determines the proper amount of time to wait for the reply of each message sent, as well as how to react when the replies do not meet the expected deadlines. Similarly to a leasing strategy (cf. Section 6.4.4.2), it can be dynamically activated at specific points in an application execution. However, a leased message protocol describes the upper bound on a set of asynchronous *message sends* within a dynamic execution rather than describing the lifetime of *objects passed* to remote parties. In this section, we describe the language constructs provided to activate leased message protocols at runtime. We defer the discussion on how to build custom leased message protocols to Section 6.7.

As shown in Table 6.1, the **`withLeaseProtocol:do:`** construct installs a given protocol within the dynamic extent of its closure argument. Listing 6.5 illustrates its usage on the running example of the context-aware music playlists. In this case, the leased message protocol employed is a *time-based leased message protocol* constructed by means of the `makeTimeBasedProtocol` helper function (line 9). This function takes as argument the time interval within the application expects to receive all the answers for the messages sent within a dynamic execution, and returns a leased message protocol object which automatically derives the upper bound of each individual messages sends from the time left in the original `maxVoteTime` interval. Note that the upper bound is relative to the time at which the message was sent to the receiver. In our example, this implies that the later a new player is discovered (lines 15-20), the shorter the upper bound of the `askToVote` message send will be. As mentioned, the activation of a leased message protocol affects the behaviour of the program within the dynamic extent of the **`do:`** closure. This means that all messages that are directly or indirectly sent within the dynamic scope of the **`do:`** closure are trapped and controlled by the protocol object. A message is said to be indirectly sent when it is sent as a result of a method invocation within the dynamic extent of **`do:`** closure. In our example, the `askToVote` message (line 5) is indirectly sent within the body of the `retrieveVote` method. Analogously to **`withLeaseStrategy:do:`**, it is possible to nest the **`withLeaseProtocol:do:`** construct which switches the protocol according to the nesting level, i.e., the innermost protocol controls the messages send within the inner **`do:`** closure, and once outside the inner layer the outer protocol is restored.

As shown in Listing 6.5 (lines 26-29), applications can react to the expiration of a leased-based message protocol by means of the **`when:expired:`** construct. We have overloaded the **`when:expired:`** construct described in Section 6.4.3 so that it can also take as first argument a leased message protocol and registers a block of code that is asynchronously trigger upon the "protocol expiration", i.e., the expiration of the

Listing 6.5: The music genre voting protocol using a time-based message protocol.

```
1  def broadcastVote(poll, maxVoteTime) {
2    def [future,resolver] := makeFuture();
3    def receivedVotes := Map.new();
4    def retrieveVote(player) {
5      when: player<-askToVote(poll) becomes: { |vote|
6        receivedVotes.put(player, vote);
7      }
8    };
9    def protocol := makeTimeBasedProtocol(maxVoteTime);
10   withLeaseProtocol:  protocol do:{
11     nearbyTeamPlayers.each: { |player|
12       retrieveVote(player);
13     };
14     def alreadySent := nearbyTeamPlayers.copy();
15     def sub := whenever: Player discovered: { |player|
16       if: !alreadySent.contains(player) then: {
17          alreadySent.add(player);
18          when: player<-getTeam() becomes: { |team|
19            if: (team == myTeam) then: {
20              retrieveVote(player);
21            }
22          }
23       }
24     };
25   };
26   when:  timeoutProtocol expired:  {
27     sub.cancel();
28     resolver.resolve(receivedVotes);
29   };
30   future
31 };
```

underlying lease associated with the given time-based leased message protocol. In our running example, **when:expired:** is employed to resolve the future returned by the `broadcastVote` function with the hashmap of received votes.

Note that the `broadcastVote` function is only expected to return the votes of friends who replied to the poll. It is not necessary to perform failure handling code when individual messages are not answered within its upper bound. However, in many cases, if the return value of a message is not received within the expected deadline, the application may need to abort the semantic protocol, or try to re-send certain messages to other receivers. To cater for these cases, we provide the **when:failedWith:** construct which takes a leased message protocol, and a block closure. When a message send fails to meet its expected delivery guarantees, the closure is triggered receiving by parameter the failed message and the original receiver of the message. As a usage example consider the following code:

```
when: protocol failedWith: { |rcv, msg|
  logger<-reportError("Voting protocol: message " + msg + "failed.");
};
```

In this example, we tell the application logger about the failure of a messages sent within the protocol. Note that the `reportError` message send may also be subject to the upper bounds defined by `protocol` if the **when:failedWith:** construct is defined within the dynamic extent of the **withLeaseProtocol:do:** construct defined at line 10. However, this message does not belong per se to the semantic protocol implemented `broadcastVote` function, i.e., it does not belong to the voting protocol. Messages can deviate from the delivery semantics defined by an active leased message protocol if they are explicitly annotated. For example, the time-based leased message protocol can be disabled for the `reportError` message by annotating it with `@OneWayMessage`. The `@OneWayMessage` is a default annotation provided by AmbientTalk to explicitly annotate messages which do not return a future.

Similarly to Listing 6.4, Listing 6.5 depicts in blue the code dedicated to deal with the timing issues of the voting protocol. Although the code is not drastically shorter than the version shown in Listing 6.4 without **`withLeaseProtocol:do:`** construct, it is important to point out that all timing assumptions are encapsulated in the protocol abstraction instead of having them scattered through the application code. As a result, the construct enhances the readability and modularity of the code. For example, if the timing assumptions need to be changed, it suffices to change the protocol code rather than all individual message send expressions.

## 6.6 Leased References vs. Leased Messages

In the previous sections, we have described the integration of the lease concept into distributed communication and computation which give rise to some useful distributed programming abstractions. This section explains their precise interaction semantics.

The lease on a leased reference specifies how long the reference can be used to communicate with the service object it refers to. A message sent via a leased reference will be received by the service object either "immediately" if the reference is connected when the message is sent[4], or at a later point in time within the lease term in case the reference is disconnected. We now describe the semantics of leased references with regard to the three kinds of asynchronous message sends featured by AmbientTalk: one-way messages, future-type messages and due-type messages.

**One-way messages.** A one-way message send does not expect a return value. Recall from Section 4.3.1 that this is actually the default semantics for an asynchronous message send in AmbientTalk. When a one-way message is sent via a leased reference, the message will be either delivered as long as the lease reference is valid, i.e., the message may be lost if it was buffered in the leased reference when the lease term expired. Hence, *we say that a leased reference bounds the delivery of a one-way message to the duration of its lease term.*

**Future-type messages.** A future-type message send expects to receive the return value by means of a future. When a future-type message is sent via a leased reference, the leased reference conceptually needs to wait for the reply. Since once the lease term expires the reference becomes terminated, it only makes sense to wait for the reply while the reference is valid. How long the leased reference should wait for the reply is thus specified by its lease term. Hence, *we say that a leased reference bounds the resolution of the message's future to the duration defined by its lease term.*

**Due-type messages.** A due-type message send expects to receive a return value within a time bound. The question again is how long the leased reference should wait for the reply. In this case, it is a bit more complex because the reference needs to combine its lease term with the one attached to the future. If the lease term of the future is shorter than the one of the leased reference, the asynchronous message

[4] In this case, immediately means that the message is directly transmitted to the actor hosting the service object, and received shortly on the receiver side. But as usual in a distributed setting, there could be some intervening of time between the sending and reception of the message due to network delays.

> is sent using the time bound stipulated by the `@Due` annotation's time interval. Otherwise, the asynchronous message is sent using the time bound stipulated by the term of the leased reference. Hence, *we say that a leased reference bounds the resolution of the message's future to the duration defined by the minimum of its lease term and the `@Due` annotation.*
>
> Note that it is possible that the message send expires on a valid leased reference because it is buffered in the message queue for a longer period than the given in the `@Due` annotation. In that case, the leased reference cancels the message transmission and the programmer needs to explicitly resend it if necessary.

In general, we will use the term *leased message* to refer to any message sent to a leased reference. If all messages sent to one service object are sent using the same timing assumptions, this can be encoded in the declaration of the leased reference on the service object (rather than encoding it repeatedly at the level of the message send using `@Due`). If at a later point, the timing assumptions need to be changed, it suffices to change one single leased reference declaration rather than all the individual message send expressions. For instance, in our running example, we could omit the `@Due` annotation on the `uploadSong` message send as it can be directly derived by above mentioned machinery incorporated in leased reference which automatically derives the time bounds on future-type messages.

Some messages may share the same timing assumptions but are not sent to the same service object. In this case, the timing assumptions do not have to be repeated at the level of each message send; this can be encoded using a leased message protocol. Usually messages sharing the same timing assumptions belong to a *semantic* protocol, such as the voting protocol described in Section 6.5.2.1. It may be some cases in which the group of timing assumptions are not part of the functionality, and thus do not belong to the same semantic protocol. Yet, expressing their timing assumptions on a leased message protocol improves the modularity, reusability and evolution of the code, as it suffices to adapt or specialize the leased message protocol (as we will shown in Section 6.7.5).

## 6.7 An Open Implementation

In the previous sections, we have described the language constructs introduced in AmbientTalk to provide leasing as a high-level referencing abstraction that materializes useful leasing management patterns. This section discusses their design and implementation. In order to provide the programmer with a referencing abstraction which is both high-level and customizable, leased references have been designed according to the rules of an *open implementation* [Kic96]. An open implementation enables programers to adjust an implementation by means of a well-defined API. We now describe leased references as an extensible AmbientTalk framework that enables programmers to add new leased reference kinds without requiring a detailed understanding of the underlying transmitter-receptor model on top of which they have been built[5].

[5] The implementation is available as a library module for the AmbientTalk/M language available at `http://code.google.com/p/ambienttalk/`.

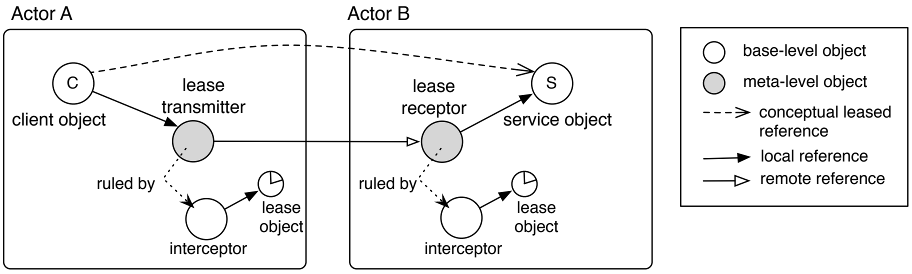


Figure 6.4: Implementation of a leased object reference.

### 6.7.1 The Leased Object Reference Framework

Figure 6.4 depicts a leased reference from the implementation point of view. Conceptually, a leased reference is a regular object reference that carries messages from a client to a service object as depicted with a dotted line. At the implementation level, a leased reference consists of a transmitter-receptor pair representing the source and target of a remote reference as explained in Section 5.2.1. The implementation of leased references relies on three adaptations of the default implementation of the transmitter-receptor pair. First, the lifetime of a remote reference is limited by means of a lease term which is initialized when the remote reference is created and associated to a client object. Second, any asynchronous message received by a leased reference is managed as described in Figure 6.3. Third, the marshalling of a leased reference has been overridden in order to ensure *pass-by-lease* semantics to a service object.

As also depicted in Figure 6.4, we expose to the programmer the different variations points of a leased reference by means of two abstractions: a `lease` object which implements methods for managing the life cycle of a leased reference, and a `interceptor` object that exposes the different variation points of a leased reference. The following subsections, we describe the two abstractions and show how programmers can modify their API to create custom leased references.

### 6.7.2 The Lease Object

A lease object has been designed along the notion of a lease described in Section 6.2 and provides methods to handle the life cycle of a leased reference. Listing 6.6 shows the API of the lease object. We categorize the API according to the two different components that comprises our notion of leases, namely a lease term and lease statements. The lease term API provides methods to activate a lease term (i.e., signal the start of the duration), get the lease period left before the term elapses, and to register listeners to react to state transitions of a lease term. The lease statement API provides methods to manage the lease term and control how leases are access by third party objects. In particular, it provides three methods (`renew`, `revoke` and `extend`) to support the three different policies for managing a lease term (described in Section 6.2.1) and the `sublease` method for specifying the lease access management policy (described in Section 6.2.2).

In order to ease the development of lease objects which provide a custom implementation to such an API, we have implemented a lease object by means of composable

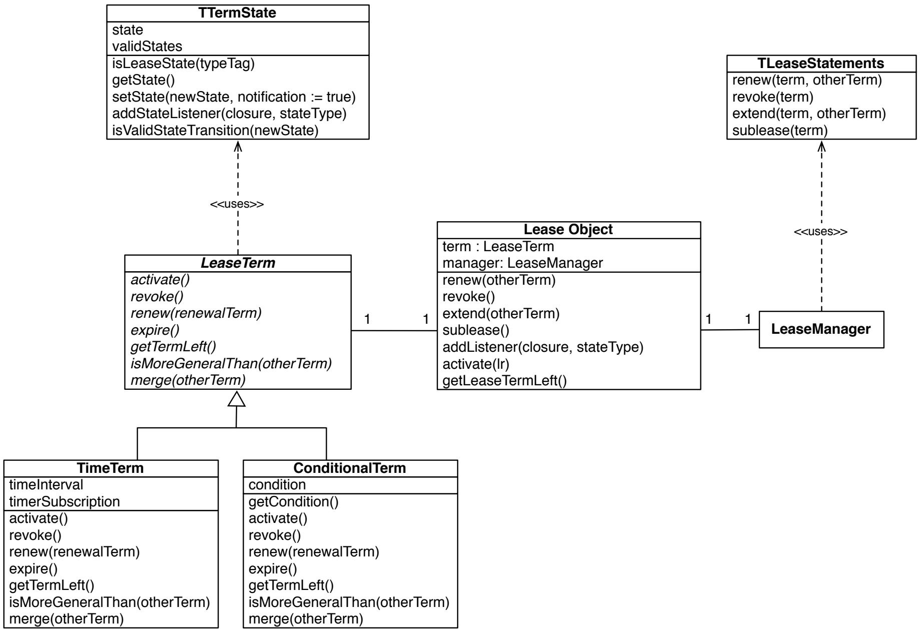


Figure 6.5: The lease object structure.

*traits*. Figure 6.5 provides a UML-like diagram depicting the implementation of leased objects. A lease object consists of two objects, a *lease term* and a *lease manager*. Actually, a lease object is just a wrapper which delegates to the lease term and manager the actual execution of the lease term API and lease statements API, respectively.

**The lease term** encodes the methods to manage the duration of a lease. The behaviour common to all lease terms is defined in a trait called `TTermState`. The trait implements the behaviour for controlling the state of a lease term and notifying observers according to the life cycle of a term according to Figure 6.3. Developers can deviate from the default behaviour by initializing the trait with a different set of states and providing a different implementation on the `isValidStateTransition` method which specifies the valid transitions among states. The `TTermState` trait is then “mixed into” into the different lease terms. By default, we provide two concrete implementations of lease terms: a time-based term that has been employed extensively

Listing 6.6: The lease object API.

```
// Lease Statements API
def renew(otherTerm);
def revoke();
def extend(otherTerm);
def sublease(otherTerm);
// Lease term API
def activate(lr);
def getLeaseTermLeft();
def addListener(closure,stateType);
def TFix := object: {
  def renew(term,renewalTerm) {
    raise: XLease.new("unable to renew a fixed-term lease" + self.print());
  };
};
def TPeriodic := object: {
  def renew(term, renewalTerm) {
    def interval := maxMethod(term.getTermLeft, renewalTerm.getTermLeft);
    term.renew(renewalTerm);
  };
};
```

TRenewalPolicy traits

```
def TNonRevocable := object: {
  def revoke(term) {
    raise: XLease.new("unable to revoke a non-canceable lease" + self.print());
  };
};
def TRevocable := object: {
  def revoke(term) {
    term.revoke(self);
  };
};
```

TRevocationPolicy traits

```
def TNonExtendable := object: {
  def extend(term, otherTerm) {
    raise: XLease.new("unable to extend a non-extendable lease" + self.print());
  };
};
def TExtendable := object: {
  def extend(term, otherTerm) {
    term.setState(activeT);
    term.activate();
  };
};
```

TExtensionPolicy traits

```
def TExclusive := object: {
  def sublease(term) {
    raise: XLease.new("unable to sublease an exclusive lease" + self.print());
  };
};
def TSublease := object: {
  def sublease(term) {
    if: (is: term taggedAs: TimeTermType) then: {
      TimeTerm.new(term.getTimeLeft());
     } else: {
       if: (is: term taggedAs: ConditionalTermType) then: {
         ConditionalTerm.new(term.getCondition());
       }
     }
  };
};
```

TAccessPolicy traits

Figure 6.6: The default lease statement policies traits.

in this work, and a first implementation of conditional terms (further discussed in Section 6.9).

**The lease manager** encodes the overall lease statements policies, i.e., provides the actual implementation of the lease statement API depicted in Figure 6.6. Rather than implementing the manager as an object providing the full implementation of the lease statement API, we represent each policy as a trait. A manager is thus represented as:

```
object: {
  import TRenewalPolicy;
  import TRevocationPolicy;
  import TExtensionPolicy;
  import TAccessPolicy;
};
```

In order to compose lease terms, `makeLease` constructor function is provided which takes a lease term object, and a lease manager returns the corresponding lease object. The code snippet below shows how it is used in the implementation of the **`lease:for:`** construct described in Section 6.4.1.

```
def lease: timeInterval for: obj {
  def manager := makeLeaseManager(TFix,TRevocable,TNonExtendable,TSublease);
  def leaseObject := makeLease(TimerTerm.new(timeInterval), manager);
  lease: obj withAgreement: leaseObject withInterceptor: TDefaultInterceptor;
};
```

The `makeLeaseManager` helper function returns the manager object resulting from importing of the four traits corresponding to the different policies depicted in Figure 6.2. Figure 6.6 shows the concrete implementations of each trait, corresponding to these policies. The `TRenewalPolicy` trait defines the `renew` method which implements the policy for renews a lease term with a specified renewal term passed as second argument. As shown in lines 6-11, the default implementation of a periodic lease provided by the `TPeriodic` trait only prolongs the lease term if the duration of the given lease term is greater than the lease term left. This avoids that if a lease will be renewed by different clients, the lease term indefinitely grows. Nevertheless, other renewal policies are definitely possible by providing a new implementation of the `renew` method. The `TRevocationPolicy` trait defines the `revoke` method which implements the strategy for revoking a given lease term. The `TExtendablePolicy` trait defines the `extend` method which implements the strategy for extending a given lease term with a specified extension term passed as second argument. The `TExtendable` trait (lines 27-32) defines the simplest extension policy; it always extends the term with the same interval as the original lease term. Note that the implementation of an extendable lease provided by the `TExtendable` trait requires a `TTermState` trait implementation that allows a transition from expired state to the initial state. As previously mentioned, this is not provided in the default `TTermState` implementation used in the current time-based and conditional terms. It needs to be encoded by providing a different implementation on the `isValidStateTransition` method. Finally, the `TAccessPolicy` trait defines the `sublease` method which implements the strategy for sharing a lease term with third party objects. As shown in lines 38-49, the default implementation of a sublease strategy provided by `TSublease` trait returns a new term whose duration is specified either by the remaining lease term for time-based terms, or with the same condition than the original lease term for conditional terms.

Before further detailing how leased references are created by means the **`lease: withAgreement:withInterceptor:`** construct, we introduce the interceptor object in the next section.

### 6.7.3 The Interceptor Interface

Our framework supports the construction of leasing variants for leased references in the form of the interceptor interface. As previously explained, leased references have been implemented as a custom transmitter-receptor pair. The interceptor interface exposes the asynchronous message process and reference marshalling protocols of the transmitter-receptor pair (explained in Section 5.2.2) at certain points of its execution. In the implementation, we represent the interceptor as a trait which provides the methods shown in Listing 6.7. The `onMessageReceived` defines the message passing semantics of a leased reference. It is called by the underlying implementation each time a transmitter or receptor receives a message. The `onReferenceCreated` and `onReferenceShared` methods, on the other hand, define the parameter passing semantics of the service object referenced to ensure a *pass-by-lease* semantics. More concretely, the `onReferenceCreated` determines how a leased reference is created on the actor hosting the client object. The underlying implementation calls it when marshalling the receptor of a leased reference. Finally, the `onReferenceShared` determines how a leased reference is passed to a third party object. It is called by the underlying implementation upon marshalling the transmitter of a leased reference.

Listing 6.7: The interceptor API.

```
def onMessageReceived(msg,lease);
def onReferenceCreated(lease);
def onReferenceShared(lease);
```

The goal of the interceptor interface is to hide as much as possible the details of the underlying implementation because the client side of a leased reference behaves slightly different than its service side counterpart. There exists two key differences between the implementation of the transmitter and the receptor of a leased reference. First, the transmitter does not grant access to a service object but to the receptor of the leased reference pointing to the actual service object (as shown in Figure 6.4). This means that the messages intercepted by the transmitter are first forwarded to the receptor of a leased reference. Before doing so, the transmitter determines the correct message delivery guarantees according to the semantics described in Section 6.6. Second, although conceptually there is only one lease object per leased reference, the implementation maintains two lease objects at each side of the reference in order to support symmetric expiration handling. This implies that the transmitter and receptor maintain their own set of **`when:expired:`** observers (which allow notification without requiring communication), and ensure that the two lease term are kept in synchronization with each other.

#### 6.7.3.1 Default Leased Reference Variants

As shown before, a leased reference created by means of the **`lease:`** construct uses the default interceptor. The code snippet below shows the message passing semantics of the default interceptor.

```
def onMessageReceived(msg, lease) {
  if: !(lease.isExpired) then: {
    self.deliver(msg);
  } else: {
    raise: XExpiredLease.new("Expired leased reference" + self.toString);
  };
};
```

If the lease object has not expired, the message is forwarded to the service object by invoking the `deliver` method. The behaviour of the `deliver` method actually depends on the side of the reference on which the `onMessageReceived` was invoked. The implementation of `deliver` in the transmitter of the leased reference first determines the message delivery semantics and then schedules the message for transmission in its message queue. The implementation of `deliver` in the receptor just forwards the message to the service object by invoking the `accept` meta method (cf. Table 5.1).

Figure 6.7 shows how such default interceptor has been extended to implement the single-call and renew-on-call variants explained in Section 6.4.2. The `onMessageReceived` method is overridden in single-call and renew-on-call interceptors to provide

```
1  def TRenewOnCallInterceptor := extend: TDefaultInterceptor with:{
2    def filter;
3    def renewalTerm;
4    def init(dftfilter := (script:{ |msg| true } carrying: []),
5      dftRenewalTerm := TimerTerm.new(lease.getInitialLeaseTerm()) {
6      filter := dftfilter;
7      renewalTerm := dftRenewalTerm;
8    };
9    def onMessageReceived(msg, lease) {
10     if: !(lease.isTerminated) then: {
11       if: (filter(msg)) then: {
12         lease.renew(renewalTerm);
13       };
14     };
15     super^onMessageReceived(msg,lease);
16   };
17 };
```

renew-on-call interceptor

```
17 def TRevokeOnCallInterceptor := extend: TDefaultInterceptor with:{
18   def filter;
19   def init(dftfilter := (script:{ |msg| true } carrying: [])) {
20      filter := dftfilter;
21   };
22   def onMessageReceived(msg, lease) {
23     if: !(lease.isTerminated) then: {
24       if: (filter(msg)) then: {
25         lease.revoke();
26       };
27     };
28     super^onMessageReceived(msg, lease);
29   };
30 };
```

revoke-on-call interceptor

Figure 6.7: Implementation of the interceptors for renew-on-call and revoke-on-call leased references.

automatic renewal and revocation of the leased object upon message reception, respectively. Both interceptors delegate the delivery of the message to the default interceptor by means of a super-send. In the case of a renew-on-call interceptor, it renews its timer before delegating the delivery of the message. As shown in line 4, the renewal time interval is the initial lease duration by default. In the case of a single-call lease, it cancels its lease upon receiving a message by calling the `revoke` method which takes care of setting the state of a lease term to terminated (in particular, revoked) so that future received messages are dropped.

Recall from Section 6.4.2 that the constructs for creating both leased reference variants can take an unary closure encoding a boolean predicate that needs to be satisfied in order for a message to renew or revoke the lease. As shown in lines 12 and 24, the interceptors only renew or revoke the lease object if the `filter` closure applies to `true`. By default, both filters are initialized to a closure that returns `true` which means that every message send renews the renew-on-call lease reference, while the revoke-on-call leased reference is revoked upon the first message send. The **`script:carrying:`** helper function allows the creation of a closure which is passed by copy in inter-actor messages copying into the closure scope the variables defined in the table given as argument. This allows us to apply renewal and revocation conditions at both sides of the reference if necessary.

#### 6.7.3.2 Adding Leased References Variants

Now that the lease object and interceptor interface have been introduced, we give further details how to compose leased reference kinds. Developing a new leased reference involves implementing an object providing the functionality for the interceptor, and passing it together with a suitable lease object to the **lease:withAgreement: withInterceptor:** construct. The following code snippet illustrates how to add the renew-on-call leased reference kind described in Section 6.4.2.

```
def renewOnCallLease: obj for: timeInterval {
  def manager := makeLeaseManager(TPeriodic,TRevocable,TNonExtendable,TSublease);
  def leaseObj := makeLease(TimerTerm.new(timeInterval), manager);
  lease: obj withAgreement: leaseObj withInterceptor: TRevokeOnCallInterceptor;
};
```

As previously mentioned, the **lease:withAgreement:withInterceptor:** construct is the leased reference creation primitive which returns a new leased reference with the properties defined by the lease object and interceptor passed as arguments. The code snippet below shows its implementation.

```
def lease: obj withAgreement: leaseObject withInterceptor: interceptor {
  def leaseReceptor :=  makeLeaseReceptor(leaseObject, interceptor);
  (reflect: obj).becomeReferencedBy: leaseReceptor;
  leaseReceptor;
};
```

The `makeLeaseReceptor` function returns a receptor in which the interceptor object is mixed into, and has been initialized with the given lease object. The **lease: withAgreement:withInterceptor:** construct installs on the service object a receptor providing leasing semantics by means of the **becomeReferencedBy:** meta method. Recall from Section 5.2.3 that the such a meta method can be called at any time during the lifetime of a service object. Note that this allows that the service object declaration does not need to be altered for leasing, promoting a separation between the distribution and failure handling aspects of the object from its functionality.

#### 6.7.3.3 Parameter Passing Semantics

In the previous section we explain how we build leasing variants by means of the provided API. The renew-on-call and revoke-on-call leased references variants alter the default message passing semantics of a leased reference thereby providing some automatic lease term management. In this section, we further describe the default parameter passing semantics of a leased reference and show how developers can create custom variants.

The code snippet below shows the parameter passing semantics of the default interceptor.

```
def onReferenceCreated(lease){
  self.transportStrategy(self, lease);
};
def onReferenceShared(lease){
  def sublease := makeLease(lease.sublease(), lease.manager);
  self.sharingStrategy(self, sublease);
};
```

When a service object is first passed to a client object, the receptor of a leased reference is marshalled instead and the system calls the `onReferenceCreated` method on the receptor. As shown in the code, the default interceptor returns the receptor itself by calling `transportStrategy` method passing the **self** variable (since the interceptor is a trait mixed into the receptor upon its creation). The `transportStrategy` method takes care of the actual marshalling of the receptor and activates the lease object. This implies that on subsequent passes of the service object to client objects, the system will hand out the same receptor. Although each client holds its own leased reference to communicate with the service object, conceptually, all client objects share the same receptor to the service object, i.e., there is a many-to-one correspondence between transmitters and receptors for a leased reference. These semantics are motivated by the trade-off that needs to be considered when creating a lease between the amount of space maintained at the machine hosting the service object, and the access rights given to the client. Using one receptor for all leased references handed out to clients reduces the amount of space that needs to be allocated. When applications require higher degrees of access control, the `onReferenceCreated` method can be overridden to, e.g., enable forms of security such as authorization, and authentication.

When a client object passes a leased reference to a third party object, the underlying implementation calls the `onReferenceShared` method on the transmitter of the leased reference. As shown in the code snippet, the default interceptor first creates a sublease of the lease object, and returns the transmitter itself with the new lease object. In our implementation, leased references employ the `TSublease` trait for detailing the subleasing policy. Recall from Figure 6.6 that the `TSublease` trait returns a new lease term with the remaining term of the original lease. Returning the same transmitter means that the third party object then receives its own leased reference which can use to communicate *directly* with the service object. The code snippet below shows how we could encode a leased reference variant which provides an *indirect* style of communication instead.

```
def TIndirectInterceptor := extend: TDefaultInterceptor with:{
  def onReferenceShared(lease){
    def sublease := makeLease(lease.sublease(),lease.manager);
    def leasedRef := lease: self withAgreement: leaseObject
          withInterceptor: TIndirectInterceptor;
    self.sharingStrategy(leasedRef, sublease);
  };
};
```

In this case, when the leased reference is passed to a third party object, the third party object obtains a leased reference to the client object which originally held the leased reference to a service object. In particular, the transmitter of the leased reference first creates a sublease of the lease object, and hands out a new leased reference by returning a new receptor in the `sharingStrategy` method. As a result, the third party object does not obtain direct access to the service object, but to the transmitter of the original client object that in turn refers to the service object. Hence, the original client object has conceptually become *sublease grantor* for the third party object. Note that combining an indirect style of addressing and subleasing results in a leased reference kind that offers higher degree of decoupling since the service object does not need to be in direct communication range of a client object to communicate with each other.

### 6.7.4 Leasing Strategies

In this section, we sketch the implementation of leasing strategies and how meta-level engineers create new leasing strategies. Recall from Section 6.4.4.2 that a leasing strategy represents a leased reference kind which is bound to each object passed to remote objects within the dynamic extent of the **`withLeaseStrategy:do:`** construct. Conceptually, a leasing strategy corresponds a first-class receptor which has not been yet installed on a service object. At implementation level, a receptor which is employed as a leasing strategy behaves slightly different from the receptors returned from the leased reference creation primitive. The main difference is that they have been altered to apply the dynamic scoping semantics of leasing strategies. This is achieved by overriding the `sendMessage` meta method of the transmitter-receptor model (described in Table 5.1).

In order for developers to use custom leasing referencing, this implementation level aspect has been factored out into the `makeStrategy` function, which serves as the leasing strategy creation primitive. The code snippet below shows its usage in the implementation of the `renewOnCall:` construct employed in the context of the music listening session code (cf. Listing 6.2).

```
def renewOnCallLease: timeInterval {
  def manager := makeLeaseManager(TPeriodic,TRevocable,TNonExtendable,TSublease);
  def leaseObj := makeLease(TimerTerm.new(timeInterval), manager);
  makeLeaseStrategy(leaseObj, TRevokeOnCallInterceptor);
};
```

This code is reminiscent of the implementation of the **`renewOnCallLease:for:`** construct previously explained, but calls the lease strategy creation primitive instead. Analogously to the leased object creation primitive, the `makeStrategy` function creates a lease strategy from the leased object and interceptor passed as arguments.

We conclude this section by briefly describing the implementation changes made to AmbientTalk for leasing strategies. In order to activate a leasing strategy within the dynamic extent of a block, we have introduced a meta variable in the actor mirror, which denotes the active leasing strategy. Recall from Section 5.3.1 that whenever an object crosses the actor boundaries, the `createReference` meta method is called, returning the receptor denoting the type of remote object reference it is passed by. The **`withLeaseStrategy:do:`** construct temporarily changes (and restores) the value of the active leasing strategy for the block of code defined by the **`do:`** closure. The object creation native (the **`object:`** construct) also has been modified to use the active leasing strategy so that any object created in the **`do:`** closure is parameter passed by leased reference as specified by the leasing strategy.

### 6.7.5 Leased Message Protocols

We conclude our implementation section by explaining the implementation of leased message protocols introduced in Section 6.5.2. A leased message protocol is an abstraction that gives control over the asynchronous messages sent within the dynamic extent of a body expression. At the implementation level, a leased message protocol is represented as a custom transmitter which intercepts asynchronous message sends. Before the actual transmission, the transmitter determines the proper upper bound for future-type message sends, and how to react when replies do not meet the expected limit. However, it behaves slightly different from the transmitter of a leased reference. In particular, it is a transmitter that is created from a leased object and a `protocol` object. A protocol object is a trait which needs to override two methods: `onMessageSend` and `onMessageSendFailed`. We describe how to build a leased message protocol by showing the implementation of the time-based leased message protocol used in voting protocol example used in Section 6.5.2.

The following code snippet shows the implementation of the `makeTimeBasedProtocol` function used for the voting protocol in Listing 6.5.

```
def makeTimeBasedProtocol(maxVoteTime) {
  def protocol:= object:{
    def startTime := now();
    def onMessageSend(rcv,msg){
          def timeLeft := maxVoteTime - (now() -startTime);
          [rcv, futurize(msg, timeLeft)];
      };
      def onMessageSendFailed(rcv, msg){
           // we do not need to abort the protocol
      };
  };
  def manager := makeLeaseManager(TFix,TNonRevocable,TNonExtendable,TExclusive);
  def leasedObject := makeLease(TimeTerm.new(maxVoteTime), manager);
  makeLeaseProtocol(leasedObject,protocol);
};
```

The makeLeaseProtocol function corresponds to leased message protocol creation primitive which returns a transmitter in which the protocol trait object is mixed into, and it is initialized with the given lease object. In this case, we create a time-based lease term representing a "timer" ( i.e., a term that cannot be renewed, revoked, extended or subleased). Each time a message is sent, the underlying transmitter calls the onMessageSend method which allows the protocol to modify the upper bound of the message before being enqueued for transmission to the service object. The futurize helper function is defined by AmbientTalk's futures module and returns a message with a @Due annotation for the given time interval. When a message expires before being transmitted, the onMessageSendFailed method is called. In this case, the onMessageExpired does not apply any failure handling code because the voting protocol did not need to be aborted if a reply was not received for a message.

Listing 6.8: Implementation of the transmitter for creating leased message protocols.

```
def makeLeaseProtocol(lease, protocol){
    extend: defaultTransmitter with: { |lease, protocol|
    import protocol;
    lease.activate(self);
    def msgObservers := Vector.new();
    def protocolObservers := Vector.new();
    def schedule(rcv,msg){
      def [originalRcv,originalMsg] := unwrapMessage(msg);
      def [rcvToSend,msgToSend] := self.onMessageSend(originalRcv,originalMsg);
      def messageWithExpiration := attachExpirationHandler(rcvToSend, msgToSend);
      rcvToSend <+ messageWithExpiration;
    };
    // called by the expiration handler when the message timer expires
    def onMessageExpired(rcv,msg) {
      self.onMessageSendFailed(rcv,msg);
      msgObservers.each: {|obs| obs<-apply()@[Control,OneWayMessage]};
    };
    //called by the lease term when it expires.
    def terminatedAccess() {
      protocolObservers.each: {|obs| obs<-apply()@[Control,OneWayMessage]};
    };
    def addExpirationObserver(code) {
      msgObservers.add(code);
    };
    def addMessageExpirationObserver(code) {
      protocolObservers.add(code);
    };
  };
};
```

Listing 6.8 sketches the implementation of the transmitter returned from the makeLeaseProtocol constructor function. The transmitter imports the protocol trait and activates the lease object so that the protocol timer starts ticking. It also maintains in two vectors storing the observers installed by means of the **when:expired:** and **when:failedWith:** constructs. When the lease term elapses, the lease term signals this by invoking the terminateAccess method which in turn notifies the **when:**

**expired:** observers. As shown in lines 6-10, the transmitter overrides the `schedule` meta method (cf. Table 5.1) to delegate to the protocol trait the calculation of the time bounds to apply to a message. Before sending the message to the receiver, it attaches an expiration handler to be able to react to the expiration of a message. Such handler will call the `onMessageExpired` method which delegates to the protocol trait the handling of the failure, and then notifies the **when:failedWith:** listeners.

Similarly to leasing strategies, we have introduced a meta variable in the actor mirror which denotes the "currently" active protocol. Recall from Section 4.6.3 that message sending is reified at the actor level by means of the `send` meta operation. We override this operation to check the value of the active protocol. If any is applied, the implementation of `send` delegates to the transmitter representing the active protocol. The **withLeaseProtocol:do:** construct changes the value of the active leasing strategy for the block of code defined by the **do:** closure. The **when:becomes:** and **whenever:discovered:** constructs have also been modified in order to apply the reset the active leasing strategy that was applied when those handlers where created.

## 6.8 Evaluation

So far we have discussed AmbientTalk language abstractions for a leasing model that adheres to the criteria distilled in Section 6.1.2. In this section, we compare AmbientTalk's support with JavaRMI [Sun98], which can be seen as state of practice mainstream technology that includes a leasing model for describing the lifetime of remote objects. Although Java RMI has not been designed for MANETs, the most representative object-oriented middleware for mobile computing, namely Jini (cf. Section 3.3.1.1), relies on Java RMI for distributed communication and computation.

We compare both approaches by means of the music player music application; our running example described in Section 6.1.1. We consider this application as a good "yardstick" because it exhibits a set of key issues that are typical in collaborative MANET applications. Its implementation in AmbientTalk, shown along Sections 6.4 and 6.5, demonstrates how developers can concisely define and manipulate leased references and how the language support eases the development of mobile applications that deal with both transient and permanent disconnections and properly reclaim their service objects. The next subsection sketches the implementation of a music player application that exhibits similar semantics in Java RMI. In doing so, we illustrate how the abstractions provided by our leasing model have to be mimicked in Java. Subsequently, we evaluate both implementations *quantitively* (based on an analysis of their lines of code) and also *qualitatively* (in the light of the criteria for a leasing model in MANETs described in Section 6.1.2). The core functionality of both implementations has been included in Appendix A[6].

### 6.8.1 An Implementation of the Mobile Music Player in Java RMI

We now describe a Java RMI implementation of music player application which exhibits similar semantics to our running example. Figure 6.8 shows a UML class diagram for this implementation. The `MusicPlayer` and `Session` classes provide methods implementing the main functionality for the music library exchange protocol depicted in Figure 6.1. The class `MusicPlayer` also includes two methods dealing with

[6] Both implementations are available at `http://code.google.com/p/ambienttalk/downloads`.

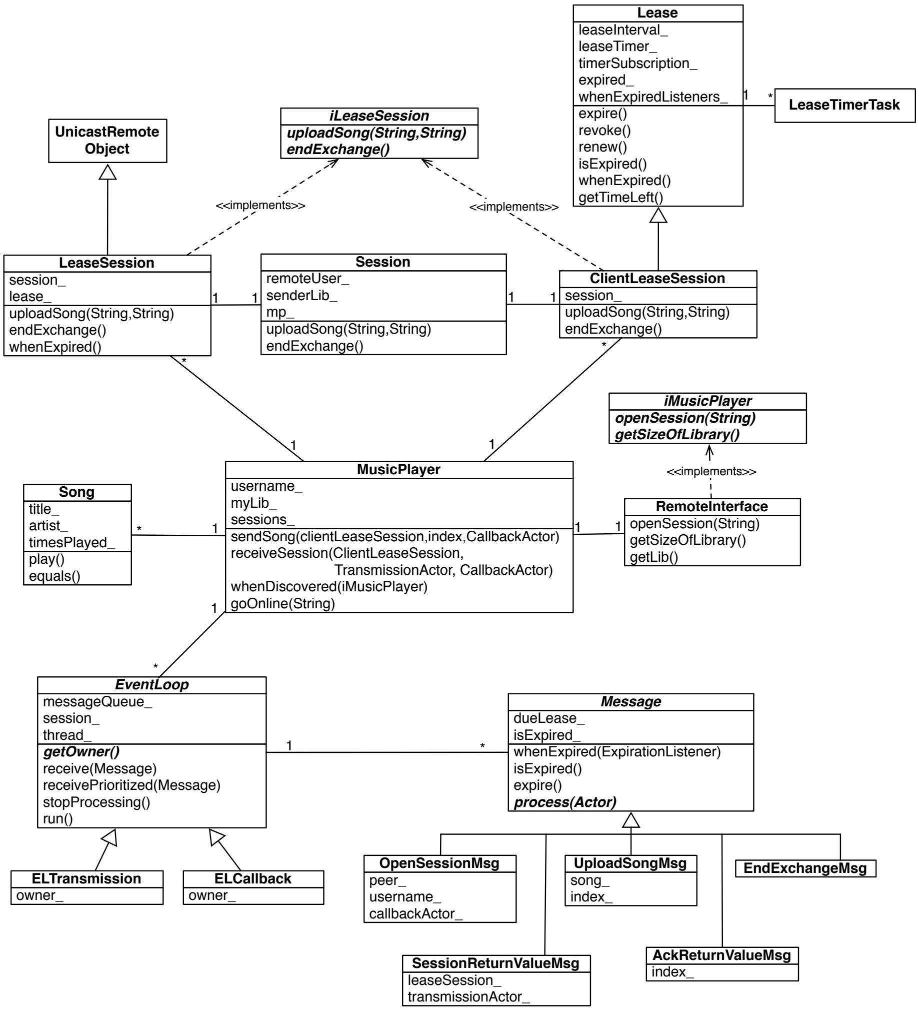


Figure 6.8: Class diagram of the implementation of mobile music player in Java RMI.

the service discovery aspect of the application: the `goOnline` exports the remote interface of the music player application to the network with a unique identifier constructed from the string passed as argument, and the `whenDiscovered` method allows the music player application to discover other music players in the network. In the implementation, we employ a predefined Java RMI registry for service discovery. It is important to notice that the focus of our comparison lies on the support for dealing with partial failures at the distributed computation and communication level, rather than service discovery. In the remainder of this section, we elaborate on these aspects.

**Distributed Communication.** Recall that once a music player establishes a session with another music player, the session should remain active as long as the exchange is active, i.e., as long as `uploadSong` messages are received. In Java RMI leases are very tightly coupled to the distributed garbage collection module. Programmers can decide on the duration of a lease by means of the `leaseValue` property which is applied to *all* remote objects in the entire Java VM. The `leaseValue` property is actually associated with the underlying socked connection timeout of a remote object reference. However, the remote reference to the remote interface of a music player application has different leasing semantics than the remote reference to a session object. In order to provide a renew-on-call leasing semantics to the session object, we employ a `LeaseSession` class. This class is mainly a wrapper for regulating access to a `Session` object depending on a `Lease` object. The `Lease` object is an ad hoc implementation of a time-based lease supporting the lease management semantics necessary for the music player application, i.e., it can be renewed and revoked. Note that the `LeaseSession` class implements the service side of a lease for a `Session` object. The `ClientLeaseSession` class implements the client side of a `LeaseSession` object, required to obtain the lease time left and place boundaries on the messages sent to the session object (described further below).

Recall from Section 6.1.1 that the application should also take care of dealing with the impact on remote references of the volatile connection phenomenon. Since Java RMI uses a synchronous communication model, the thread executing a remote method call blocks upon a network disconnection. This would make the application unresponsive to discovery events notifying new music player peers in the network. To solve this issue, two different threads need to be spawned: a *transmission* thread that performs a remote method invocation, and a *callback* thread that awaits the return values. These threads need to communicate with each other by means of *message* objects which wrap a remote call. In addition, the transmission thread must ensure that messages are buffered during a network disconnection. In our implementation, the `EventLoop` class wraps a thread and a message queue and provides methods to make a thread send a remote message. The `ELTransmission` and `ELCallback` classes represent the aforementioned transmission and callback threads.

**Distributed Computation.** We implemented five message classes corresponding to the various remote messages shown in Figure 6.1. Each `Message` subclass needs to implement the `process` method which is in charge of performing the corresponding remote method invocation. Recall from Section 6.5.1 that `uploadSong` and `openSession` have different delivery guarantees because the return value of the `uploadSong` message is only useful if it is received within the remaining duration of the session. Since in Java RMI *every* remote method invocation has the same timeout (the one denoted by the lease associated to a remote object reference), we implemented this by associating a `Lease` instance with a `Message` (stored in the `dueLease_` field). The code snippet below shows the `process` implementation for the `OpenSessionMsg` message subclass:

```
public void process(EventLoop a) throws NoSuchObjectException {
  try {
    iLeaseSession session = peer_.openSession(username_);
    if (!expired_){
      dueLease_.revoke();;
      callbackEL_.receive(new SessionReturnValueMsg(
        session, (ELTransmission) a));
      System.out.println("session opened");
    }
  } catch (NoSuchObjectException e0){
    throw e0; // trying to use an expired session, throw again.
  } catch (RemoteException e) {
    //exception while session active, reschedule to simulate buffering.
    if (!expired_){
      System.out.println("unable to open a session - reschedule the message");
      a.receivePrioritized(new OpenSessionMsg(
        peer_, username_, timeout_, callbackEL_));
      e.printStackTrace();
    }
  }
}
```

The process method is called by the ELTransmission thread when the OpenSessionMsg object reaches the head of its message queue. When process is invoked, a remote method invocation for openSession is performed on the RemoteInterface of the MusicPlayer instance. Note that the ELTransmission thread serving this message is blocked until the openSession invocation finishes, or the lease associated to the OpenSessionMsg object expires. The result of openSession is encoded by the SessionReturnValueMsg message. When the callback thread processes a SessionReturnValueMsg message, an ClientLeaseSession object is created for the received session object.

In order to react to the expiration of the such a lease, the Message class provides the whenExpired method which takes as parameter a listener to be applied when the lease expires. We also modified the ELTransmission class so that the transmission thread only attempts to perform remote method invocation for messages which have expired in its message queue. The code snippet below shows the usage of the whenExpired method in the constructor of the OpenSessionMsg class.

```
dueLease_.whenExpired( new ExpirationListener(){
  public void leaseExpired() {
    expired_ = true;
    SessionReturnValueMsg expiredMsg = new SessionReturnValueMsg(null, null);
    expiredMsg.expire();
    callbackEL_.receive(expiredMsg);
    System.out.println("TIMEOUT openSession ");
  }
});
```

The expiration listener places an empty expired message in the callback threat to signal that we do not need to wait anymore for the result of the remote method invocation, and notifies the user of the expired message.

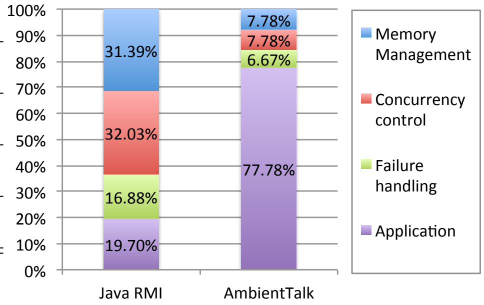
### 6.8.2 Quantitative Evaluation

Thanks to the language constructs presented earlier in this chapter, the implementation of the music application in AmbientTalk counts merely 90 loc while its implementation in Java RMI counts no less than 462 lines. Table 6.9a summarizes the lines of code for both implementations according to four different concerns: 1) memory management: includes the code to setup a renew-on-call lease for the music session and reclaim the used data structures upon lease expiration, 2) concurrency control: includes the code to ensure the responsiveness of the application in the face of transient disconnections, 3) failure handling: includes the code to have time-based delivery policy guarantees on remote messages and 4) application-level code. Note that the code for service discovery has not been taken into account in this comparison because, as previously mentioned, we focus our discussion on the distributed communication and computation aspects of the application. Incorporating the aspect of service discovery would only make the comparison worse for Java RMI since AmbientTalk's features a built-in publish/subscribe service discovery mechanism.

To further compare the differences between both implementations, we depict in Figure 6.9b the percentages of lines of code for each implementation according to

| | Java RMI | AmbientTalk |
|---|---|---|
| Memory management code | 145 | 7 |
| Concurrency control code | 148 | 7 |
| Failure handling code | 78 | 6 |
| Application-level code | 91 | 70 |
| **Total lines of code** | **462** | **90** |

(a) Summary of the lines of code



(b) Chart of the percentages

Figure 6.9: Overview of the lines of code (a) and % (b) for the music player application according to four concerns: memory management, concurrency control, failure handling and application-level code.

the four concerns. The most notable difference is the application concern. While the application-level code has a similar magnitude in both implementations, the graph shows that most of the code in AmbientTalk consist of application-level code (77,78%) while in Java RMI this is less than 20% (19,70%). This is mainly because, leasing in Java RMI is very tightly coupled to the distributed garbage collection module, and it has not been integrated with a decoupled communication model at all. As a result, the Java RMI implementation required to encode a number of patterns by hand which we discuss in the next section.

### 6.8.3 Qualitative Evaluation

In Section 6.1.2, we postulated three criteria to which a good leasing model for MANETs should adhere. In the light of these criteria, we now evaluate the language support for ambient-oriented leasing introduced in AmbientTalk and Java RMI.

**Criterion#1: Leasing an Intermittent Connection** JavaRMI remote references *do not support time decoupling directly*. Time decoupling needs to be manually encoded by means of two separated threads to asynchronously perform remote method invocations and receive the results. In addition, the transmission thread must ensure that messages are buffered while the reference is disconnected. This is not required in AmbientTalk as leased references themselves abstract away from the connection state. Client objects can send messages via a leased reference as long as it is not expired, independently of the state of the connection, because a leased reference buffers messages while disconnected. When knowledge of to the network connectivity is required, developers can place dedicated listeners that are triggered whenever the leased reference becomes disconnected and reconnected. As such, leased object references combine leasing with asynchronous communication into one coherent language concept that deals with both transient and permanent failures.

**Criterion#2: Leasing Management Patterns** Java RMI provides leases to delimit the lifetime of remote object references. However, it is *not possible to specify lease periods on a per-application, per-class or per-object basis* because the duration of leases is controlled by a VM property (`leaseValue`). This property is actually associated with the socked connection timeout. As such, there is no difference between a lease associated to a remote reference and a timeout for a message send. Upper bounds on remote method invocations need to be explicitly encoded in terms of timers. In addition, a transmission thread needs to take into account the message expirations.

It is important to remark that Java RMI leases provides a built-in leasing pattern for managing *connected* remote object references. As long as there is no network disconnection, a remote reference is considered in use. The client-side runtime system transparently renews the lease to a remote object implicitly once the `leaseValue` reaches half of its value. This may lead to unnecessary network traffic since control messages are sent to keep a remote object alive which may not be used anymore. To signal that a client stopped using a remote reference, developers have to explicitly make sure to clear local references to the remote reference. For example, in the music player application, code was added to stop the transmission and callback threads and to remove the session from the hashmap storing active opened sessions. If clients do not add this code, the remote object cannot be collected unless there is a disconnection exceeding the `leaseValue` timeout.

In AmbientTalk, we designed leasing language abstractions that gives developers fine-grained scoping mechanisms to decide which leasing agreement should be used when and where:

- Leased references allow developers to specify leasing agreements in a *per-object* basis.

- Due-type message sends allow developers to specify different leasing agreements on a *per-message* basis.

- Leased references and due-type messages have been complemented with two *dynamic scoping abstractions*. Leasing strategies and leasing protocols allow developers to specify leasing agreements to a group of objects and messages, respectively. Leasing strategies the alleviate burden of expressing leasing in a per-object basis, enabling a group of objects to be passed to client objects under the same leasing agreement. When a group of messages share the same timing assumptions, these timing assumptions do not have to be repeated at the level of each message send; this can be encoded using a leased message protocol.

Moreover, AmbientTalk decouples the notion of network connectivity from failure handling. As such, a leased reference becomes expired even if a it was still connected when its lease term elapsed.

**Criterion #3: A Customizable Leasing Framework** The Java RMI leasing model *is not designed to enable programmers to deviate from the default leasing behaviour*. The system provides an interface to the distributed garbage collection module based on so-called *dirty* and *clean* calls. However, such interface is not meant to be used by application programmers. Leasing patterns such as the renew-on-call pattern used by the music application needs to be implemented explicitly, requiring repetitive renewal code at client and server-side. In AmbientTalk, useful leasing management patterns have been made available in the form of dedicated leased references such as the renew-on-call and revoke-on-call leased references. Since leased references have been designed as an open implementation, meta-level engineers can build custom leasing variants not provided in the default implementation.

**Criterion#4: Symmetric Expiration Handling** *No notification is performed in Java RMI upon lease expiration.* Client objects are notified about the expiration of a lease only if they issue a remote method call on an expired lease (which throws an exception). At server side, the `unreferenced` method is called on a remote object only when *all* clients disconnect. In the music player application, the registration of listeners with leases had to be explicitly encoded. In AmbientTalk, dedicated language constructs are provided for the registration of expiration listeners at both sides of a leased reference. This enables communicating parties to schedule appropriate compensating actions when their logical connections terminates.

## 6.9 Limitations and Future Work

We now discuss some limitations of our leasing abstractions and give some hints on how we think they can be addressed in future work.

**Service Discovery** A limitation of our current implementation is that leased reference have not been integrated with Ambientalk's service discovery constructs. In order to limit the time that objects are *explicitly* exported on the network as featured by some service discovery abstractions like Jini's lookup service, developers must declare an object with a lease before calling the **`export:as:`** construct (cf. Section 4.4.2). We would like to provide an extension to the **`export:as:`** construct to ease the export objects with a lease. However, this requires changes on the underlying language kernel. This is a technical limitation derived from the fact that the AmbientTalk kernel language employs itself an event-loop concurrency model to implement all concurrent activities such as AmbientTalk virtual machines. Since service discovery has been modeled as a separate event loop within an AmbientTalk VM, the serialization and deserialization of explicitly exported objects currently do not trigger as expected the object marshalling protocol (cf. Section 4.6.3) on which leased references rely.

**Conditional Leases** As we mentioned in Section 6.2, a term that lasts until a specific event occurs is called a *conditional* term. An event can be described in the programming language model as a method call on the service object, or a predicate defining application-specific conditions. Although our implementation includes a conditional term object (as explained in Section 6.7.2), we have not yet studied the integration of conditional terms in leased object references. Note that since a conditional term is not associated with a specific time limit, the duration is not certain at the time of the grant. Consequently, the meaning of lease term management changes. For example, it is not evident what it means to renew, extend, or sublease a conditional term since its length is not definite. It would be interesting to study whether and how our scoping abstractions can be applied to conditional terms.

**Leasing Management Patterns** In order to ease selecting a suitable lease duration, we explored leased references as an extensible open implementation in which several leasing management patterns are provided and allows meta-level engineers to encode custom ones. Other research has focused on analytical models for comparing lease protocols and determining the optimal lease duration [BRL01, DST03, VZKS11]. In this context, some interesting ideas have been proposed to adapt the lease duration based on different conditions. For example, in [DST03], Duvvuri et al. describe three strategies to determine the lease duration based on the object access pattern, and server space. More concretely, *age-based leases* provide a leasing strategy in which the server grants short leases to frequently modified objects and long leases to long lived ones, while *renewal frequency-based leases* implement a strategy in which the server grants longer leases to "popular" objects (object that are frequently accessed). Finally, a third strategy is provided by *state space overhead-based leases* in which the lease duration is set according to the number of valid leases granted for a particular object. An interesting topic of future work would be to enlarge the set of default leasing patterns to offer such adaptive techniques. Their implementation is definitely possible since our underlying transmitter-receptor model allows meta-level engineers to monitor accesses and modifications on the referenced objects (by means of the asynchronous message process protocol described in Section 5.2.2), and to control the remote references being created within an actor (by means of the actor's reference management protocol described in Section 5.3.1.1). However, it remains to be investigated whether and how the interceptor API described in Section 6.7 needs to be modified in order to accommodate such leasing variants.

**Leasing Strategies** As previously mentioned, leasing strategies employ a similar scoping mechanism than the one found in context-oriented programming [CH05]. Similar to ContextL layers, our leasing strategies can be activated and deactivated at runtime. While activation and deactivation of layers is thread-local in ContextL, our leasing strategies are local to an actor. ContextL allows developers to dynamically compose layers by nesting the `with-active-layers` construct within the control flow of a program. In contrast to ContextL, two leasing strategies cannot be simultaneously active at the same time for a given actor. The main reason for this is that, in our context, a leasing strategy represents the parameter passing strategy for a group of objects, rather than behavioral variations of arbitrary code (expressed as partial class definitions). The composition of leasing strategies is more problematic than layer composition in context-oriented programming languages because it is more prone to raise conflicting behaviour. This could be alleviated by introducing a mechanism in which developers could express dependency relationships among leasing strategies (e.g., to allow or disallow certain leasing strategies compositions). Composition mechanisms for leasing strategies remains a relevant open issue.

Scoping mechanisms similar to the ones supported in context-oriented programming have attracted considerable attention in the aspect-oriented community. For example, the CaesarJ [AGMO06] language supports dynamic deployment of aspect on thread-local scope. Tanter in [Tan08] identifies the problem of existing scoping models for dynamic aspect deployment as a lack of control over propagation of an aspect both along call stack and delayed evaluation, as well as deployment-specific join point filters. Our leasing strategies get propagated on the "call stack" (which in our work corresponds to the history of asynchronous message sends), but they do not get applied to certain delayed evaluation. According to Tanter, delayed evaluation propagation determines the propagation of an aspect in a closure or object created in the lexical scope of the aspect body. While our **`withLeaseStrategy:do:`** construct can capture object created within its scope, a lease strategy is not engrained to closures so later closure applications are out of its reach. This is a technical limitation of our software platform (shared also with leased message protocols) due to the fact that AmbientTalk does not allow to modify closure creation reflectively, requiring changes in the kernel interpreter. However, we have modified the most relevant delayed computation constructs for distribution (**`when:becomes:`** and **`whenever:discovered:`**) to be able to reset the active leasing strategy or leased message protocol when those handlers are applied.

Filtering objects exported within the **`withLeaseStrategy:do`** block can be achieved in our approach by explicitly deactivating the leasing strategy at all required locations. To address the identified issue, Tanter introduces the notion of *deployment strategies* which expresses the scoping semantics of a dynamically-deployed aspect in a parameterized way by means of three components: two propagation functions to specify whether an aspect propagates along the call stack and within delayed evaluation, and a join point filter to filter the joint points seen by the aspect. An interesting point of future work would be to bring ideas of such a parameterized approach into leasing strategies in order to provide fine-grained control over scoping of leased references.

## 6.10 Notes on Related Work

Before concluding this chapter, we describe the most closely related leasing models, and we highlight the lease-based approaches which have influenced the design of our leasing model. The notion of lease has been applied to distributed computing to solve a number of problems such as consistency maintenance, resource management, and memory management. In this work, we have mostly combined ideas of both lifetime management and resource management approaches. Moreover, it exposes leases as first-class values.

### 6.10.1 Consistency Maintenance

Leases were first introduced by Gray et Cheriton in the context of distributed file cache consistency protocols to provide mutual exclusion in an efficient and fault-tolerant fashion [GC89]. In this context, a lease grants control over writes to a certain datum during the term of the lease. In order to read a datum, a client first needs to acquire a lease. When a client writes a datum, the server invalidates the cached datum of all clients whose leases have not expired, and postpones the write until it obtains the leaseholders approval or until the leases expire. In order to read the datum after the lease expires, a client must first ask the server to renew the lease before it can access the datum. Leases provide strong consistency semantics by notifying active clients (clients holding a lease) when a datum changes. However, they may incur a considerable overhead as each datum has an associated lease and *all* its updates must be notified.

Several extensions have been proposed to make leases scalable for maintaining consistency in web proxy caches by allowing a server to grant a lease for a group of data on the server [YADL99, NKS$^+$02]. A cooperative lease belongs to a group represented by a *leader* that is responsible for managing all leased-based operations with the server, e.g., renewal of the individual object leases, and propagating updates on an object in the group to clients that are caching it. Interestingly, different policies for lease renewals can be applied to cooperative leases. An eager renewal policy implies that the leader renews the lease upon expiration until it is notified by the client not to do so. A lazy renewal policy, on the other hand, does not renew the lease but sends a "lease expired" messages to all clients within the group which can then request renewal when the object is accessed. Our renew-on-call lease variant is similar to eager renewals, but renewal does not happens after lease expiration but while the lease is active upon message sending on a leased reference.

Most current leasing models stem from the original proposition by Gray et Cheriton in which the notion of a lease is coupled with a global time and assumes synchronized clocks. Interestingly, asynchronous leasing [BDG02] proposes an implementation of leasing where *physical* time is replaced by *logical* time (represented as a interval of integers). Leasing is then implemented with a quorum-based algorithm in which a lease is represented by a *shared* object exporting a `lease(startInterval, endInterval)` function. Upon invoking this function, a process sends a `LEASE` message to all processes, and returns true if it receives an `ACK_LEASE` message from the majority of processes. Each processes contains a permission tree representing the time interval and acknowledges a lease request according to two properties: (1) a lease is acquired at most once for a given interval (called *at most leasing property*), and (2) a lease cannot be refused if no lease has been requested for an overlapping interval (called *lease mandatory property*). In spite of the fact that the algorithm has been designed for an asynchronous system, it is not suitable for a mobile setting because it assumes a crash-stop failure model (i.e., processes fail by crashing and do not recover from a crash).

## 6.10.2 Memory Management

Leases have also been used as a technique for memory management to describe the lifetime of remote objects in a fault-tolerant fashion. In particular, they have been incorporated in some distributed object-oriented systems including Java RMI [Sun98], and .NET Remoting [MWN02] to enable the reclamation of objects in the face of failures of client objects. We now further describe how these leasing models related to our approach in the light of the criteria distilled in Section 6.1.2.

### 6.10.2.1 Java RMI

We have already extensively discussed Java RMI in Section 6.8. To sum up, leases in Java RMI are tightly coupled to the distributed garbage collector (DGC) which is based on the algorithm introduced by Birrell for network objects [ABW93, BNOW93]. More concretely, Birrell proposed a distributed reference listing algorithm which included a time-based mechanism to detect process failures and update the reference tables accordingly. Each process periodically pings the clients that hold references to its objects; if the ping is not acknowledged after a certain amount of time, the client is assumed to have terminated, and objects that it referred to may become candidate for garbage collection. Java RMI adopted this algorithm employing the “lease” terminology instead. Java RMI’s leases are built into a synchronous communication model (based on RPC) which does not decouple objects in time or synchronization [EFGA03]. As such, they are only be used to reclaim unused connected remote references.

### 6.10.2.2 .NET Remoting

The .NET Remoting framework incorporates leasing in combination with the concept of *sponsorship* for managing the lifetime of remote objects [Low03]. Sponsors are third-party objects which are contacted by the framework when a lease expires in order to check whether or not that party is willing to renew the lease. Clients can register a sponsor on a lease and thus decide on the lifetime of server objects.

Similar to JavaRMI, the .NET Remoting framework leases are used to reclaim unused connected remote references. In contrast to the low-level primitives offered in

JavaRMI, the .NET Remoting framework incorporates a leasing pattern at the heart of its design. Leases are automatically extended on every call on the remote object by the time specified in the `RenewOnCallTime` property. If that property is not set, lease renewal can be achieved by registering a sponsor. Variations on the integrated pattern need to be built on top of sponsor and lease interface abstractions. The lease interface provides methods for overriding some leasing properties (e.g. `RenewOnCallTime`), renewing the lease, and the registration of sponsors.

Expiration handling is not provided in the .NET Remoting framework. Although the system does indeed contact sponsors upon lease expiration, there are no guarantees that the system will contact the sponsor of a specific client as it may ask several sponsors until it finds one willing to renew the lease.

### 6.10.3 Resource Management

In [JK00], Jain and Kircher described the concept of leasing as a software design pattern for simplifying resource management. We now discuss a number of object-oriented systems that follow such a practice of applying leases as a general abstraction for resource management with regard to the criteria introduced in Section 6.1.2.

#### 6.10.3.1 CORBA

Alesky et al. present in [AKS05] a service-based approach to introduce the concept of leasing into the CORBA specification. In order to provide reusable leasing functionality for different CORBA-based applications (independent of their programming language), leasing is modeled as a dedicated CORBA service. In this context, a lease denotes the right of a *resource claimant* to use a resource for a limited period of time, and offer methods to renew and revoke its duration. A resource can be practically any CORBA entity as long as it implements two methods (used by leases) to start and stop the use of the resource.

The authors integrate two types of leasing patterns depending on the type of resource claimant. First, *observed* claimants receive a lease which observes the claimant so that if it terminates, the lease gets cancelled. The object is periodically queried to detect if it is still alive. Due to the volatile connections phenomenon, such leases do not seem appropriate for a mobile setting: the claimants may only be disconnected temporarily, causing the lease to be cancelled erroneously. Second, *notified* resource claimants receive a lease which notifies the claimant as soon as it expires. The lease is then automatically renewed once at server side to give the claimant sufficient time to renew the lease if necessary. Other leasing patterns may be possible but need to be built on top of the above mentioned architecture, e.g., automatic renewal of leases can be accomplished by making explicit renew calls on the lease interface.

To the best of our knowledge, expiration handling is only supported at the client side using notified claimants.

#### 6.10.3.2 Jini

Jini has been already discussed in Section 3.3.1.1 as a framework built on top of Java which allows clients and services to discover and set up an ad hoc network. We now focus our discussion on its leasing model. Jini introduces leasing to deal with unannounced disconnections of clients and services within the federation. In particular, services advertise themselves in the lookup service for a particular duration determined by the registration leased handed out by the lookup service itself. Services must be explicitly renew their lease and if they cannot, the lookup service will remove the service advertisement upon lease expiration.

In order to ease renewal operations, Jini provides a data structure for the systematic renewal of a set of leases. Leasing patterns can be built using such a lease renewal service which can act as a intermediary implementing the protocol to communicate with the remote service. Expiration handling can be achieved at client side by registering an event listener on a lease renewal set. When the lease is about to expire, an expiration warning event is generated notifying all registered listeners for that set.

In [BMR03], Bowers et al. propose an interesting extension to Jini's leasing model for varying lease periods in response to the system size. In particular, the authors consider two self-adaptive algorithms that enable a Jini system to change the lease duration in order to guarantee a minimum average responsiveness of a Jini's lookup service. One algorithm restricts lease term based on bandwidth consumption of the leasing system to provide best responsiveness of the as the system size varies. The second algorithm inverts the process of granting a lease by making the lookup service periodically poll on a multicast channel from which lease holders listen. A poll includes the duration over which the lookup service will listen for leaseholders to respond, and the time increment over the given duration when the next poll can be expected. If the leaseholder does not respond within the broadcasted duration, the lookup service cancels the lease.

Finally, as previously mentioned, Jini relies on the synchronous communication model of Java RMI. Although Jini's architecture is flexible enough to accommodate a leasing model integrated with a decoupled communication model, to the best of our knowledge, Jini does not implement this functionality.

## 6.11 Conclusion

In this work, we propose to devise a failure handling model for MANETs that is centered around the concept of leasing. At the beginning of this chapter, we identified a number of criteria to be exhibited by a leasing model to be used in a MANET setting. We require a leasing model that (1) takes into account the volatile connections phenomenon, (2) provides different leasing patterns to manage the lifetime of leases, (3) provides facilities to enable experienced developers to construct new custom leasing variants, and (4) allows both lease holder and grantor to react to and schedule clean-up actions upon lease expiration. Subsequently, we proposed a leasing model which incorporates the concept of a lease as a first-class object that exposes a number of methods that can be overridden, enabling meta-level engineers to express new kinds of leasing agreements among distributed processes.

We integrated the lease concept into AmbientTalk's distributed communication model giving rise to the abstraction of a *leased object reference*: a time-decoupled object reference which deals with both transient and permanents failures. We designed leased references as an extensible framework which integrates useful patterns, e.g., renew-on-call and single-call leased references, and provides a well-defined API on which custom leased reference variants can be built.

We have also integrated the lease concept into AmbientTalk's distributed computation model giving rise to *due-type messages*. Due-type message enable developers to steer the message delivery process, when the default timing assumptions provided by a leased reference are not suitable for individual messages.

Finally, we have investigated novel language abstractions to alleviate the programming effort that leasing introduces in the form of *leasing strategies* and *leased message protocols*. Leasing strategies enable developers to assign the same leasing semantics to a group of *objects* passed to remote parties within a dynamic execution extent of a code block. Leased message protocols allow programmers to abstract the timing assumption of a group of *messages* into a separate abstraction, avoiding that they need to be manually specified for each message send.

In the next chapter, we investigate the applicability of ambient-oriented leasing into a data-driven model, namely, the tuple space model.

# Chapter 7

# Ambient-Oriented Leasing for Tuple Spaces

Until now we have mainly focused on combining leasing with decoupled communication in a distributed object-oriented model. However, data-driven models like tuple spaces and publish/subscribe also exhibit some forms of decoupled communication interesting for a mobile environment as discussed in Chapter 3. In this chapter, we make a first integration of ambient-oriented leasing into tuple spaces (described in Section 3.3.2). While the incarnation of ambient-oriented leasing into a distributed object-oriented model explained in the previous chapter builds upon the decoupled communication model present in AmbientTalk, the embodiment of ambient-oriented leasing for tuple spaces entails *both* the design and implementation of a tuple space model, and the incorporation of leasing into the model. This has led us to propose a novel tuple space approach called TOTAM ("Tuples On The Ambient") inspired by TOTA (cf. Section 3.3.2.2). We describe how TOTAM enables an ambient-oriented programming style by integrating leasing for tuple management in the face of partial failures. We first motivate our work, and then describe the TOTAM tuple space model (in Section 7.2) and its concrete implementation in AmbientTalk (in Section 7.3). We demonstrate the applicability of our model by using it in a non-trivial mobile peer-to-peer application (in Section 7.4) and provide an operational semantics for our model (in Section 7.5). TOTAM's design and implementation is attributed to the author together with Christophe Scholliers [SGD09, SGBDMD10].

## 7.1 Motivation

TOTAM has been designed with the goal of integrating a failure handling model (adhering to the criteria distilled in Section 2.4) for the tuple space paradigm. We motivate the need for TOTAM based on a number of software engineering issues exhibited by prior tuple space models targeting the mobile environment. We focus our discussion on two aspects: (1) how they support a decoupled communication model, and (2) how they deal with network failures and the extremely dynamic context to which MANETs expose applications.

**Decoupled Communication.** Most tuple space-based approaches targeting the mobile environment follow either a the *federated* tuple space model (including LIME

[MPR01], TuCSon [OZ99], and EgoSpaces [JR04]), or a *replication-based* model (such as TOTA [MZ04] and [MP06]). Recall from Section 3.3.2 that, in a federated tuple space model, the set of tuples accessible for a device consists of all the tuples that each device in the vicinity carries in its tuple space. Devices can post and read tuples from such a federated tuple space by means of the traditional tuple space operations, and non-blocking operations are provided to read tuples by means of *reactions*: callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space. In this model, however, time decoupling is sacrificed in order to guarantee atomicity for tuple removal operations since tuples can only be exchanged when the communication partner that issued a tuple space operation is in range of the device offering the requested tuple. In contrast, in a replication-based model, tuples are replicated among collocated devices in order to increase data availability in the face of intermittent connectivity. In TOTA's model, tuples are also equipped with a *propagation rule* that prescribes how a tuple is to hop from one tuple space to another one. These propagation rules provide programmers with a flexible mechanism to express more elaborate kinds of tuple sharing than simple merging of tuple spaces as proposed by LIME. Tuples can be thus exploited to achieve context-awareness based not only on connectivity but also on semantic information. Removal of tuples, however, needs to be manually encoded in terms of the propagation rules; as such atomic removal of tuples is not guaranteed by the underlying system. Nevertheless, atomicity for removal operations is an essential feature to support coordination between applications. In addition, since tuples are transmitted to all communication partners, information cannot be hidden or scoped. By transmitting tuples potential malicious or non-intended users may be provided with sensitive information, as well as increasing the burden on network traffic.

**Context-Awareness.** Current tuple spaces developed for the mobile environment promote a style of coordination that abstracts away from the mobility and distribution of devices in the environment, easing the development of MANET applications. However, this may be too restrictive for certain applications requiring higher degrees of context-awareness [MLE02]. For example, context information can be used to optimize application behaviour given the scarce resources of mobile devices. To deal with this issue, LIME extends traditional tuple space operations with a *location* that allows users to post tuples annotated with the tuple's intended destination tuple space [MPR01]. However, this trades decoupling in space and the anonymous communication style of tuple spaces for more control over the underlying system configuration (since developers can obtain details about the nodes present in the network and use them to specify the tuple space in which a tuple should be placed).

In general, context information in tuple spaces is represented by the ability to *read* certain tuples from the environment. However, this representation is inappropriate and can even lead to an erroneous perception of context in both federated and replication-based tuple space models. The former ties the perception of context to network connectivity which does not always yield the expected result because the visibility of the tuples (and thus context) depends on collocation of devices holding these tuples. The latter causes context to be perceived even if a device has left that context a long time ago because a permanent disconnection leads devices to perceive they are in the application's context forever. In short, the main issue is that the ability to read a tuple from the environment does not give any guarantees that the context information carried by the tuple is appropriate for the reader. This forces programmers to manually verify that a tuple is valid for the application's context situation after the tuple is read. As the complexity of applications increases, it becomes impractical for programmers to explicitly maintain a view of the application context and adapt it accordingly as the environment changes.

While many approaches allow to monitor the connection status of the devices forming part of the underlying network using dedicated tuples, very few of them provide a high-level representation of failures. The most notable exception is Tiamat [ME03], a federated model which includes a leasing model to allow programmers to specify upper boundaries on the availability of tuples in the tuple space.

## 7.2 TOTAM: Ambient-oriented Programming with Tuple Spaces

In this work, we introduce TOTAM, a novel tuple space model that gathers concepts from both federated and replication-based tuple spaces, and extends them with a number of features to overcome the aforementioned limitations (i.e., an ambient-oriented programming style with a leasing model, a scoping mechanism, and context rules), and offer a coherent tuple space-based programming model for MANET applications. In the remainder of this section, we sum up the most important features of our model.

**The Core Model** The model underlying TOTAM tuples extends the notion of a traditional tuple space with machinery to control the scope and visibility of tuples. Figure 7.1 depicts our model. A device in the network corresponds to a virtual machine (VM) carrying one or more TOTAM tuple spaces. Each virtual machine forms a *TOTAM system*. TOTAM systems are interconnected by means of a MANET, forming a *TOTAM network*. The composition of a TOTAM network varies with to the changes on the network topology as devices move about.

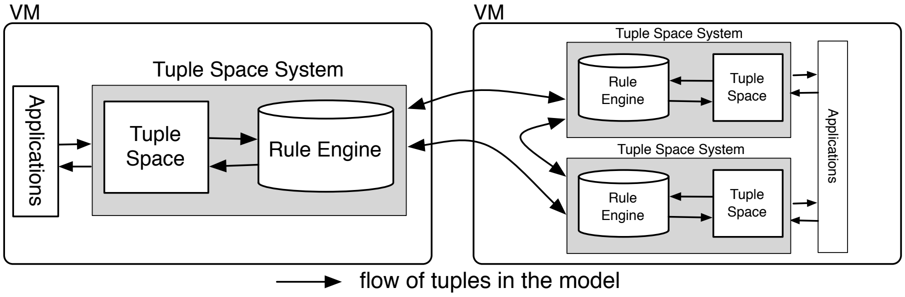


Figure 7.1: The TOTAM tuple space model.

A TOTAM system consists of a tuple space, and a rule engine which infers when a tuple should be perceived by applications. The tuple space serves as the interface between applications and the TOTAM system. As an alternative to blocking operations, we provide the notion of a *reaction* to a tuple (similar to LIME reactions [MPR01]): applications can register an observer that is asynchronously notified when a tuple matching a given template is read or removed from the tuple space.

The tuple space of a TOTAM system contains two types of tuples. *Public tuples* denote tuples that are shared with remote TOTAM systems, and *private tuples* denote tuples that remain local to the tuple space in which they were inserted and thus will not be transmitted to other TOTAM systems. Applications can insert private and public tuples in the tuple space by means of the `out` and `inject` operation, respectively. As in LIME, applications can access tuples coming from the network without knowing the different collocated TOTAM systems explicitly.

**Distribution of Tuples in the Network** When two TOTAM systems discover each other in the network, the public tuples contained in their tuple spaces are cloned and transmitted to the collocated TOTAM system according to a *propagation protocol* (reminiscent of TOTA propagation rules). However, in contrast to TOTA, TOTAM triggers the propagation protocol *before* a tuple is being physically transmitted to a new TOTAM system, and *after* it is received by the new TOTAM system. Thus, the tuple itself contains all the information needed to dynamically adjust its scope in the TOTAM network while being propagated. This differs from techniques in which the tuple space itself is scoped and tuples have to be inserted in a certain scope which can not be changed after the facts. In addition, such a scoping mechanism avoids unnecessary exchange of tuples and enhances privacy because the protocol can enforce that a tuple is not transmitted to a TOTAM system which is not in its scope.

To be able to compute the scope of a tuple, each tuple space has a *tuple space descriptor*. This descriptor contains semantic information that is used by the tuples at sending time in order to decide whether a certain TOTAM system is in their scope. Descriptors are exchanged between two TOTAM systems when they meet for the first time or whenever a system decides to change its description.

**Coordination** Our model combines replication of tuples for read operations (to support time-decoupled communication) while guaranteeing atomicity for removal operations (to support synchronization). In order for a remove operation to succeed, the system which created and injected the tuple in the network (called the *originator* system) needs to be connected. The underlying model does not remove tuples unless the originator system has acknowledged the operation. As such, a remove operation in our approach is executed atomically as defined in Linda [Gel85]: if two processes perform a remove operation for a tuple, only one removes the tuple. When an originator is asked to remove one of its (stored) tuples by another system, it removes the tuple and injects an *antituple* for the removed tuple in the network. By means of antituples, TOTAM systems can “unsend” tuples injected to the network. For every tuple there is (conceptually) a unique antituple with the same format and content, but with a different sign. All tuples injected by an application have positive sign while their antituples have a negative sign. Whenever a tuple and its antituple are stored in the same tuple space, they *annihilate* one another, i.e., they both get removed from the tuple space.

**Supporting Context-awareness** TOTAM tuples can carry a *context rule* that describes the runtime conditions under which the tuple is visible to an application, facilitating the development of context-aware applications deployed in a mobile environment. More precisely, a context rule is conceived as a set of conditions defined in terms of the presence of other tuples in the receiving tuple space. Such context rule is defined by the creator of the tuple and gets transmitted together with the tuple when the tuple is injected in the network. Defining context rules in terms of tuples allows the application to abstract away from the underlying hardware while keeping the simplicity of the tuple space model (both interactions and context information are defined using tuples).

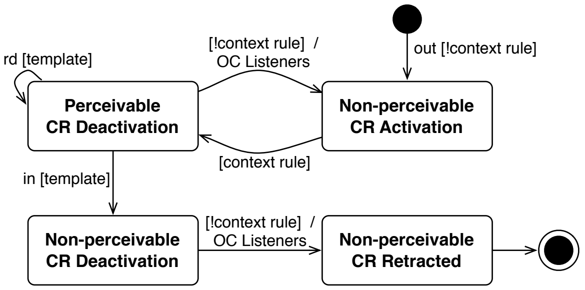

rd [template]
[!context rule] / OC Listeners
out [!context rule]
Perceivable CR Deactivation
Non-perceivable CR Activation
[context rule]
in [template]
Non-perceivable CR Deactivation
[!context rule] / OC Listeners
Non-perceivable CR Retracted

Figure 7.2: Lifespan of a context-aware tuple

When a tuple is inserted at a certain TOTAM system, the tuple is first handed over to the rule engine which installs the necessary machinery to evaluate the tuple's context rule. When the rule engine infers that the conditions on a context rule are satisfied, the tuple's context rule is triggered and the rule is said to be *satisfied*. Only when the context rule of a tuple is satisfied, is the tuple inserted in the tuple space of the TOTAM system. At that moment, the applications are able to read the tuple.

The rule engine plays a central role in our model as it takes care of reflecting the changes to the receiver's context so that applications cannot perceive those tuples whose context rule is not satisfied. In particular, it observes the insertion and removal of the tuples in the tuple space to infer which context rules are satisfied, and subsequently control which tuples present in the tuple space should be actually accessible by applications. As a result, programmers no longer need to infer the presence or absence of tuples manually as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

**The Lifespan of a Context-aware Tuple** Context rules introduce a new dimension to the lifespan of a tuple. Not only can a tuple be inserted or removed from the tuple space, it can now also be perceivable or non-perceivable for the application. Figure 7.2 shows a UML-state diagram for the lifespan of a *context-aware tuple*, i.e., a TOTAM tuple carrying a context rule. When an application inserts a tuple in a TOTAM system[1], the tuple is non-perceivable and its context rule is asserted by the rule engine. The rule engine then starts listening for the activation of that context rule (CR activation in the figure).

A tuple will become perceivable depending on whether or not the context rule is satisfied. If the context rule is satisfied, the tuple is perceivable and it is subject to tuple space operations (and thus becomes accessible to the application). If the tuple is non-perceivable, the tuple is not subject to tuple space operations but its context rule remains in the rule engine. Every time a tuple's context rule is not satisfied, the out-of-context listeners of a tuple (OC listeners in the figure) are triggered. Applications can install listeners to be notified when a tuple moves out of context.

Upon performing an `in` operation, the tuple is removed from the tuple space but its context is not modified. As such, the tuple is considered not to be perceivable (as it is no longer in the tuple space) and its context rule remains in the rule engine. Once out of the tuple space, the rule engine listens for the deactivation of the context rule. Once

[1] To keep the figure concise `out` denotes the insertion of a private or a public tuple.

the context rule is no longer satisfied, the context rule is retracted from the rule engine, and the tuple will be eventually garbage collected.

**A Best Effort Leasing Model** In order to support resource management in the face of permanent disconnections, we integrate our leasing model into public tuples. *Designing a leasing model for a replication-based model, however, is challenging because tuples get replicated through a highly dynamic network topology.* The main problem is that at the moment a tuple needs to be removed it might not be possible to reach all systems where it was replicated to. As a result, ensuring global consistency across the network may be impossible to attain without incurring communication overhead, or making assumption about the mobility of devices [MP06].

In TOTAM, we have introduced a leasing model which does not guarantee strong consistency, but only guarantees *best effort*. The underlying system tries hard to remove all replicas of a tuple, but does not give guarantees when/if this will happen. It is important to notice that the system does guarantee that the tuple is only removed once. All tuples are injected in the network with an associated lease denoting the total lifespan of a tuple. It is determined by the application that creates and injects the tuple to the network. Programmers can specify the lease time interval by means of dedicated support (explained later).

A lease is encoded as part of the propagation protocol, and as such, it is transmitted together with the tuple when it gets replicated and transmitted to another TOTAM system. When the lease term has elapsed, independently of the state in which a tuple is, the tuple becomes candidate for garbage collection in the TOTAM system. This means that the tuple's context rule is retracted from the rule engine, and the tuple is removed from the tuple space if necessary.

The transitions for leasing have been omitted from Figure 7.2 to keep it clear and concise. The lease of a tuple adds a transition from any state to the state representing that a tuple is non-perceivable and its context rule is retracted. When a public tuple gets removed as a result of an `in` operation, the underlying system sends an antituple to those systems that have previously received a replica of the removed tuple. If a TOTAM system cannot be reached, the removal of the tuple is delayed until its lease expires in the disconnected TOTAM system, or until it is in range with one of the TOTAM systems carrying its antituple.

## 7.3 Programming in TOTAM

In this section we describe TOTAM from a programmer's perspective. We have implemented TOTAM as a middleware in AmbientTalk[2]. Table 7.1 summarizes TOTAM's programming API which we further detail in this section. We illustrate the set of operations provided by TOTAM by means of a concrete application called *Guanotes* that we use as running example throughout this section.

### 7.3.1 Running Example: the Guanotes Application

Guanotes is one of the default applications of *Urbiflock* [GBLCS$^+$11]: a framework implemented in AmbientTalk for the development of mobile social applications run-

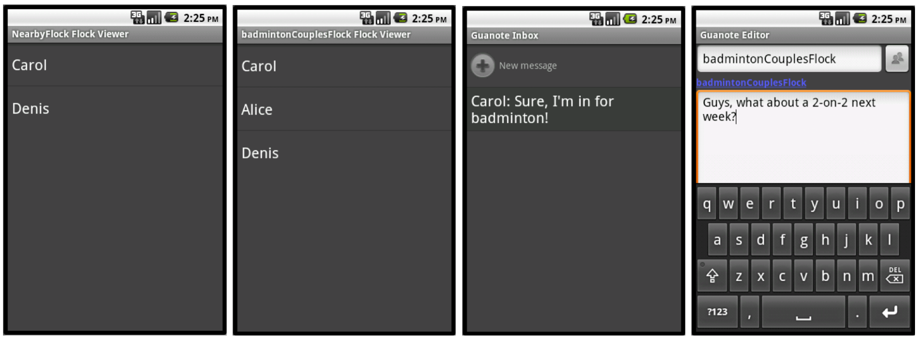
[2] TOTAM is available in the AmbientTalk standard library which can be downloaded with the language at `http://soft.vub.ac.be/amop`

| | |
|---|---|
| *Middleware operations* | |
| makeTupleSpace(descriptor) | creates a tuple space with the given descriptor. |
| **tuple:** content<br>**withPropagationProtocol:** protocol | creates a tuple with the given list of fields. The optional parameter **withPropagationProtocol:** denotes the propagation protocol by which the tuple may be injected in the network. |
| **propagationProtocol:** closure | returns a protocol object which extends the default propagation protocol with the given code block. |
| **var:** symbol | return a variable from the given symbol. |
| *Tuple space operations* | |
| goOnline | publishes the tuple space in the TOTAM network. |
| rdp(template) | returns a tuple matching the template or **nil** if none is present at the time of invocation. |
| rdg(template) | returns all tuples matching the template or **nil** if none is present at the time of invocation. |
| out(tuple, lease) | adds a private tuple to the tuple space. |
| **inject:** tuple **inContext:** rule<br>**withLease:** interval | adds a public tuple to the tuple space. **inContext:** and **withLease:** optional parameters allow to specify the associated context rule and (lease) time interval, respectively. It returns a publication object with methods to stop the tuple's propagation, and retract it from the network. |
| **when:** template **read:** closure<br>**outOfContext:** oocClosure | registers a reaction on the tuple space for the given template. When a tuple matching the template is available in the tuple space, the closure listener is applied binding all variables of the template to the matching tuple. Optionally, the **outOfContext:** closure can be specified to react when the matching tuple is no longer perceivable. |
| **when:** template **in:** closure<br>**outOfContext:** oocClosure | works analogously to **when:read: outOfContext:** operation, but registers a reaction on the removal of a tuple matching the template. |
| **whenever:** template **read:** closure<br>**outOfContext:** oocClosure | works analogously to **when:read: outOfContext:** operation, but triggers the closure listener for *every* perceivable tuple matching the template. |
| **whenever:** template **in:** closure<br>**outOfContext:** oocClosure | works analogously to **when:in: outOfContext:** operation, but triggers the closure listener for *every* perceivable tuple matching the template. |

Table 7.1: Programming API of TOTAM

ning on Android phones (inspired by Facebook). It allow end-users to interact with their social networks (organized in *flocks*) by means of messages. A flock typically denotes a group of nearby users that match a number of user-defined criteria. Messages in Guanotes are called *guanotes*. A guanote can be sent to the users belonging to a flock which are currently in their direct neighbourhood, or to all (reachable) users belonging to the target flock regardless of whether they are currently connected in the Urbiflock platform.

To exemplify the kinds of interactions using Guanotes, consider the following scenario: Alice and Bob are in the cafeteria of the university sports complex when they decide it would be nice to play some badminton. Since reserving the badminton field is rather expensive, Bob decides to invite some extra players by taking his mobile phone and using Guanotes to send a message to the couples *currently* in the neighborhood who like to play badminton. Luckily, Carol and Denis who also wanted to play badminton see the invitation. They reply to Bob's message whereafter they meet and start playing a game. After the game, they get the wild idea to organize a badminton competition for next week. Again Bob takes his mobile phone and decides to send an invitation to *all* couples at the university who like badminton.



Carol

Denis

Carol

Alice

Denis

Carol: Sure, I'm in for badminton!

badmintonCouplesFlock

badmintonCouplesFlock

Guys, what about a 2-on-2 next week?

Figure 7.3: Bob's (a) NearbyFlock, (b) badmintonCouplesFlock, (c) Guanotes inbox and (d) Guanote editor

Figure 7.3 shows Urbiflock on Bob's device during the process of typing a guanote after playing a match with Carol and Denis. In particular, four different screenshots are displayed: (a) the contents of Bob's `NearbyFlock`, (b) the contents of Bob's `badmintonCouplesFlock`, (c) Bob's Guanotes inbox (that contains a guanote previously received from Carol replying to his first invitation), and (d) the editor for a new guanote to invite all his friend's couples to participate in a badminton tournament. Note that there are few users collocated at this time with Bob (people in the `NearbyFlock` shown in (a)) which belong to the `badmintonCouplesFlock` (b). As he would like to reach all couples interested in badminton for the tournament, he selects the `badmintonCouplesFlock` in the receiver list in Figure 7.3(d).

### 7.3.2 Defining and Inserting TOTAM Tuples

In order to create a TOTAM system and add it to the network, programmers can invoke the `makeTupleSpace` constructor function as follows:

```
def totam := makeTupleSpace(descriptor);
totam.goOnline();
```

This operation initializes a TOTAM system including the rule engine and the tuple space. The newly created TOTAM system is then published in the TOTAM network by means of the `goOnline` operation. This operation returns a publication object with a `cancel` method to be able to remove the system from the TOTAM network. From then on, the newly created system will perpetually look for other systems in the TOTAM network and exchange its descriptor with them.As explained before, the descriptor contains semantic information relevant to the propagation of tuples. In the Guanotes application, the description is embodied in the user profile which is implemented as an isolate including fields representing the semantic information associated with the user (e.g., name, gender, hobbies, etc.).

We provide the `out(tuple)` operation to insert a private `tuple` in the tuple space. In order to insert a public tuple, thereby making it available to other collocated TOTAM systems, the **`inject:`** operation is provided. The code excerpt below illustrates the injection of the guanote depicted in Figure 7.3(d).

```
def aGuanote := tuple: [guanote, username, badmintonCouplesFlock,
  "Guys, what about a 2-on-2 next week?"];
totam.inject: aGuanote;
```

A tuple is created by means of the **`tuple:`** operation which takes as argument a list of fields (implemented in AmbientTalk as a table). As usual, the first field of a tuple is its type name, i.e., a type tag (cf. Section 4.2). In this case, the tuple's type name is the `guanote` type tag and a guanote consists of the sender of the tuple (stored in the `username` variable), the receiver target group (in this case the `badmintonCouplesFlock` group) and a textual message.

In its simplest form, the **`inject:`** operation takes as argument a tuple and injects the tuple into the network with a *default context rule* (i.e., the context rule is always true), and a *default lease* (i.e., a lease time interval predefined at the actor level). It returns a publication object with two methods: a `cancel` method that allows programmers to stop the propagation of the tuple in the TOTAM network, and a `retract` method that allows developers to "unexport" a tuple injected into the network (by making the system send an antituple to all TOTAM systems which received that tuple). A tuple carries a default propagation protocol that propagates its to all reachable TOTAM systems, exactly as in TOTA, i.e., it does not restrict the propagation of tuples. We explain how to encode and apply custom propagation protocols in Section 7.3.4.

Table 7.1 shows the complete form of the **`inject:`** operation. The **`withLease:`** parameter takes a time interval denoting the specific duration of the lease associated with the tuple. The **`inContext:`** parameter takes a context rule defined as a table containing the set of templates and constraints that need to be satisfied for the tuple to be perceivable. Constraints are conceived as logical conditions on the variables used in a template. For example, consider that certain guanotes are only visible if the user is in, e.g., the cafeteria area. In order to model that a device is in a room, the application could inject a dedicated public tuple as follows:

```
totam.inject: (tuple: [inRoom, cafeteriaRoom])
inContext: [tuple: [location, ?loc], withinBoundary(roomArea,?loc)];
```

The `inRoom` tuple is a "helper" tuple which allows a guanote to determine when the user is in the cafeteria area. To this end, the `inRoom` tuple carries a context rule which consists of two terms that need to match:

- First, there must be a tuple in the tuple space encoding the location of the user, i.e., matching the **tuple:** [location,?loc] template[3]. The ? operator indicates a variable in a template. As shown in Table 7.1, variables are actually defined using the var: construct which takes a symbol as argument. This chapter will use the ? operator in order to ease the reading. In our example, the template matches *any* location tuple in the tuple space.

- Second, the location tuple needs to satisfy a constraint: its coordinates have to be within the area of the room. The withinBoundary function returns such a constraint given the coordinates stored in the ?loc variable and the cafeteria area stored in roomArea variable.

### 7.3.3 Reading and Removing Tuples from the TOTAM Network

In order to read and remove public tuples from the TOTAM network, programmers can use reactions to register a block of code that is executed when a tuple matching a template is inserted in the tuple space, reminiscent to LIME reactions. In what follows, we describe the 4 kinds of reactions supported (shown in Table 7.1). The **when:read:** operation takes as argument a template to observe in the tuple space, and a closure that serves as a event handler to call when the tuple matching the template is available in the tuple space. It actually performs a reaction to a read operation, i.e., the matching tuple is not removed from the tuple space. In its simplest form, the **when:read:** operation only triggers the event handler once for a matching tuple. If several perceivable tuples match the template, one is chosen non-deterministically. The **whenever:read:** operation works analogously but it triggers the event handler for *every* perceivable tuple matching the template. The code excerpt below illustrates the usage of **whenever: read:** in the implementation of the Guanotes application.

```
def listenForGuanotesToOwner(guiListener) {
  def guanoteTemplate := tuple: [guanote, ?from, ?to, ?msg];
  totam.whenever: guanoteTemplate read: {
    guiListener<-guanoteReceived(from, to, msg);
  };
};
```

The listenForGuanotesToOwner function is called from the application GUI in order to start listening for tuples whose type name is guanote. When a perceivable tuple matches the template, the **read:** closure is applied binding all variables of the template to the values of the matching tuple. In this example, the application just extracts the information from the tuple and sends it to the GUI listener to be displayed[4].

Finally, the **when:in:** and **whenever:in** operations work analogously to the previous ones but, they perform a reaction to a removal operation rather than a read operation. In other words, these operations remove the tuple from the tuple space before applying the closure block. Recall that if the tuple to be removed comes from another TOTAM system, then the underlying TOTAM system has to contact the originator TOTAM system to atomically remove the original tuple. If that removal fails, the replicated tuple is not removed from the local tuple space and the closure is simply not triggered.

Our approach extends a LIME reaction with the notion of *context*: the event handler for a reaction can only be triggered when the tuple matching the pattern is perceivable.

[3] A template is created by means of the **tuple:** operation as well. However, only templates can take variables as fields.

[4] In the actual implementation, the tuple contents are converted into an AmbientTalk object which can be understood by the Java GUI using AmbientTalk's interoperability layer.

The complete forms of the previously described operations allows developers to react to a tuple moving out of context by installing an `outOfContext` listener. In the code snippet, the **`whenever:read:outOfContext:`** operation expresses that certain guanotes are only visible from within a certain room.

```
def inroomTemplate := tuple: [inRoom,?name];
totam.whenever: inroomTemplate read: {
  guiListener<-display("You are in room" + name);
} outOfContext: {
  guiListener<-display("You moved out of room" + name);
};
```

In this case, each time an `inRoom` tuple is matched, the application detects that the user moved in a certain room. Once the user leaves the room, the `inRoom`'s context rule is no longer satisfied and the `outOfContext` closure is applied. In short, the extended versions of `when(ever):*` operations asynchronously apply the first closure when the tuple is perceivable, and the **`outOfContext:`** closure when the context rule of the matching tuple is not satisfied.

Apart from the reactions previously explained, we also provide the `rdp` and `rdg` operations to read one and all tuples in the tuple space, respectively, that match a given template (at the point in time when the operation is executed).

### 7.3.4 Writing Application-Specific Propagation Protocols

As previously mentioned, tuples hop from one TOTAM system to another according to a propagation protocol carried by the tuple itself. A propagation protocol is encoded in TOTAM as an isolate object implementing a number of methods which are called by the the underlying TOTAM system. It basically adapts TOTA propagation rules API (shown in Listing 3.2) for controlling the propagation of tuples *before* and *after* transmitting a tuple. In this section, we explain the API of TOTAM's propagation protocols and how programmers can define custom propagation protocols.

Listing 7.1 shows the protocol before propagating the tuple (i.e., at sender side). First, `decideDie` is called. It returns a boolean indicating whether or not the tuple should be removed. If the protocol decides not to remove the tuple, the protocol will be asked whether a potential receiver is in the scope of this tuple. `inScope` takes as argument the descriptor of the sender and receiver tuple space. The protocol can then decide to propagate the tuple to the receiver descriptor. Finally, before transmitting the tuple, the protocol can change its content (line 5).

Listing 7.2 shows the protocol on the receiver side. A tuple is first asked whether it should enter the receiving tuple space (line 1). It will then be allowed to execute some action on the local tuple space (line 2). Finally the tuple can change its content

Listing 7.1: The TOTAM propagation protocol at sender side.

```
1  if: tuple.decideDie(tupleSpace) then: {
2    remove(tuple);
3  } else:{
4    if: tuple.inScope(senderDescriptor, receiverDescriptor) then: {
5      tuple := tuple.changeTupleContentOnSend();
6      transmit(tuple,locationOf(receiverDescriptor));
7    }
8  };
```

Listing 7.2: The TOTAM propagation protocol at receiver side.

```
1  if: tuple.decideEnter(ts) then: {
2    tuple.doAction(ts);
3    tuple := tuple.changeTupleContentOnReceive();
4    if: tuple.decideStore(ts) then: {
5      ts.out(tuple);
6    };
7    tuple
8  } else: { nil };
```

and decide to notify the local tuple space that it arrived (line 3 and 4). Tuples with a protocol which does not notify the local tuple space can be used e.g., to remove inappropriate tuples.

The code excerpt below illustrates how the Guanotes application creates a custom propagation protocol to be carried by guanotes which should only be received by users within direct communication range, i.e., systems only one hop away.

```
1  def makeGuanoteToNearbyUsers(from, msg){
2    def oneHopProtocol := propagationProtocol: {
3      def hops := 0;
4      def inScope(senderTs, senderDes, receiverDes) {
5        hops < 1;
6      };
7      def changeTupleContentOnReceive(ts) {
8        hops := hops + 1;
9        self.getContent();
10     };
11   };
12   tuple: [guanote, from, to, msg] withPropagationProtocol: guanotesProtocol;
13 };
```

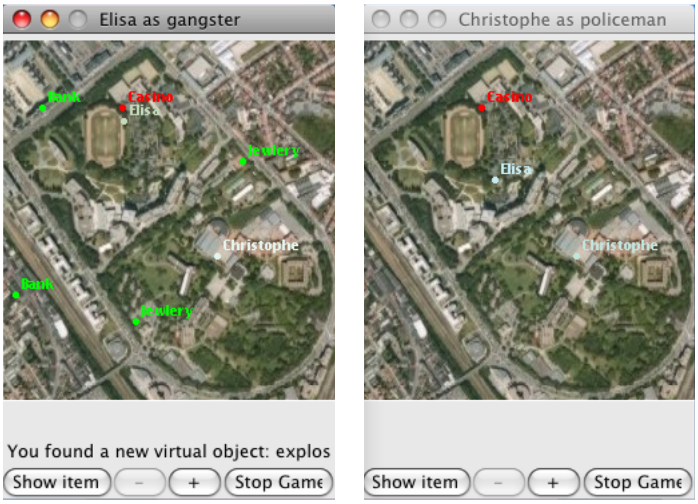
**`propagationProtocol:`** is an operation that allows the programmer to quickly create a custom protocol. It expects as argument a code block that is used to extend the default propagation protocol prototype (which corresponds to a tuple which always propagates to every tuple space encountered). In this example, the `inScope` and `changeTupleContent` methods are overriden in order to limit the propagation of the guanote. The `inScope` method will verify that the tuple has been transmitted only one hop. The `changeTupleContent` increments the hop counter. Line 12 shows programmers can associate a propagation protocol with a tuple by means of the **`tuple:withPropagationProtocol:`** construct.

## 7.4 Case Study: Flikken

We demonstrate the applicability of TOTAM's programming API by showing the implementation of *Flikken*[5]: a game in which players equipped with mobile devices interact in a physical environment augmented with virtual objects. Flikken is a significant subset of an augmented reality game inspired by the industrial game "The Target"[6]. The game consists of a dangerous gangster on the loose with the goal of earning 1 million euro by committing crimes. In order to commit crimes the gangster needs to collect burglary tools around the city (knives, detonators, etc). Policemen work together to shoot the gangster before he achieves his goal.

[5] Flikken means *cops* in Flemish.

[6] `http://www.lamosca.be/en/the-target`



You found a new virtual object: explos

Figure 7.4: Flikken GUI on the gangster device (left) and a policeman device (right).

Figure 7.4 shows the gangster's as well as a policeman's mobile device at the time the gangster has burgled the local casino. The gangster knows the locations with larger amounts of money (banks, casinos, etc). When a gangster commits a crime, policemen are informed of the location and the amount of money stolen. Policemen can see the position of all nearby policemen and send messages to each other in order to coordinate their movements. The gangster and policemen are frequently informed of each other's positions and can shoot at each other.

Flikken epitomizes MANET applications that react to context changes in the environment. Examples of such changes include player's location, appearance and disappearance of players, and the discovery of virtual objects while moving about. Moreover, how to react to these changes highly depends on the receivers of the contextual information. For example, virtual objects representing burglary tools should only be perceived by the gangster when he is close to their location while they should not be perceived by policemen at all. In what follows, we describe the coordination and interaction between policemen and the gangster using tuples.

### 7.4.1 Design

Every player has a TOTAM system. Once the game starts, policemen and gangster communicate by means of the TOTAM network. Throughout the city various context providers (i.e., TOTAM systems) are placed playing the role of virtual objects or crime locations by broadcasting the necessary tuples. A special type of context provider is at the headquarters (HQ) of the players which signals the start of the chase.

In this section, we only describe the set of tuples coordinating the core functionality which counts 11 tuples (6 of which carry a custom context rule, and 2 a custom lease) and 7 reactions. Table 7.2 shows an overview of the tuples used in the game. The tuples are divided into five categories depending on the entity that injects them in the environment, i.e., all players, only gangster, only policemen, headquarters and city context providers. A tuple is denoted by the term $\tau$ and the first element of a tuple

| Tuple Content | Tuple Context Rule / Lease | Tuple Description |
|---|---|---|
| All Players | | |
| $\tau$(TeamInfo, uid, gip) | [true] | Private tuple denoting the player's team. |
| $\tau$(PlayerInfo, uid, gip, location) | [$\tau$(TeamInfo, ?u, ?team), ?team $\neq$ gip] / [6 min] | Injected to opposite team members every 6 minutes to notify the position of a player. Location is a 2-tuple indicating the (GPS) coordinates of the player. |
| $\tau$(OwnsVirtualObject, GUN, bullets) | [true] | Private tuple inserted by players when they pick up their gun at their HQ. |
| Only The Gangster | | |
| $\tau$(CrimeCommitted, name,location,reward) | [$\tau$(TeamInfo, ?u, POLICEMAN)] | Notifies policemen that the gangster committed a crime. |
| $\tau$(OwnsVirtualObject, type, properties) | [true] | Private tuple inserted when the gangster picks up a virtual object in the game area. |
| Only Policemen | | |
| $\tau$(PlayerInfo, uid, gip, location) | [$\tau$(TeamInfo, ?u, gip)] / [1 min] | Notifies the position of a policemen to his colleagues every time he moves. |
| Headquarters | | |
| $\tau$(InHeadquarters, location) | [$\tau$(PlayerInfo,?u,?team,?loc), inRange(location, ?loc)] | Notifies that the player entered his HQ. Used to start the chase (when this tuple moves out of context for the gangster's HQ) and to reload policemen's guns. |
| $\tau$(CrimeTarget, name, location) | [true] | Notifies the gangster of the position of crime targets. |
| $\tau$(CommitCrime,name, location,reward,vobj) | [$\tau$(PlayerInfo,?u, GANGSTER,?loc), inRange(location,?loc), hasVirtualObjects(vobj)] | Notifies the gangster of the possibility of committing a crime. `hasVirtualObjects` takes an array of virtual object ids and checks that the gangster has the required `OwnsVirtualObject` tuples. |
| City Context Providers | | |
| $\tau$(VirtualObject, id, location) | [$\tau$(TeamInfo, ?u, GANGSTER), $\tau$(PlayerInfo,?u, GANGSTER,?loc), inRange(location, ?loc)] | Notifies the gangster of the nearby presence of a virtual object. `inRange` is a helper function to check that two locations are in euclidian distance. |
| $\tau$(Rechargeable-VirtualObject,GUN, BULLETS) | [$\tau$(InHeadQuarters,?loc), $\tau$(OwnsVirtualObject,GUN,?bullets), ?bullets $<$ BULLETS ] | Represents the player's gun. The gangster gets only one charge at the start of the game, while policemen's guns are recharged each time they go back to their HQ. |

Table 7.2: Overview of the Tuples used in Flikken

indicates its type name. We use capitals for constant values.

The TOTAM system on each player's device is identified by a descriptor including its username and the team that he belongs to. However, since there is communication between policemen and the gangster, tuples injected by players are not scoped on a team basis. We make use of context rules to control the perception of certain tuples when necessary (e.g., to make sure the gangster does not see certain tuples). In contrast, all tuples injected by the TOTAM system at the headquarters are scoped on a team basis using the player tuple space descriptor.

The player's TOTAM system carries a private tuple $\tau$(TeamInfo, uid, gip) indicating which team he belongs to. Every player injects his location to the TOTAM network by means of the tuple $\tau$(PlayerInfo, uid, gip, location). This tuple is injected into the tuple space with a custom lease as we will explain later. Both the `PlayerInfo` and `TeamInfo` tuples are often used in other tuple's context rules to identify the current whereabouts of a player and his team. For example, the tuple representing a grenade uses these tuples as follows.

```
totam.inject: (tuple: [VirtualObject, grenade, location]);
inContext: [tuple: [TeamInfo, ?u, GANGSTER],
            tuple: [PlayerInfo, ?u, GANGSTER], ?loc],
            inRange(location, ?loc) ]
```

The tuple $\tau$(VirtualObject, grenade, location) should be only visable if the receiver is a gangster whose location is physically proximate to the virtual object. The `inRange` function checks whether the gangster location (given by `?loc` in the `PlayerInfo` tuple) is in euclidian distance with the location of the grenade (stored in `location`). Upon removal of a `VirtualObject` tuple, a private tuple $\tau$(OwnsVirtualObject, object) is inserted in his tuple space. `CommitCrime` tuples notify the gangster of a crime that can be committed. As crimes can only be committed when the gangster has certain burgling items, the context rule of the `CommitCrime` tuple requires that certain `OwnsVirtualObject` tuples are present in the tuple space. For example, in order for the gangster to perceive the `CommitCrime` tuple for the `grandCasino`, a $\tau$(OwnsVirtualObject,grenade) tuple is needed as shown below.

```
totam.inject: tuple(CommitCrime, grandCasino, location, reward)
inContext: [tuple: [TeamInfo, ?u, GANGSTER],
            tuple: [PlayerInfo, ?u, GANGSTER, ?loc],
            inRange(location, ?loc),
            tuple(OwnsVirtualObject,grenade)];
```

Note that since context rules can be developed separately, this enables programers to reuse rules to build different kinds of tuples, increasing reusability. As shown above, we have reused the `inRange` function to build both the rule for the different `VirtualObject` tuples and `CommitCrime` tuples. Decomposing a tuple into content and context rule also leads to separation of concerns, increasing modularity.

Each player also registers several reactions to (1) update his GUI (e.g., to show the `OwnsVirtualObject` tuples collected), and (2) inject new tuples in response to the perceived ones. For example, when a gangster commits a crime, he injects a tuple $\tau$(CrimeCommitted, name,location,reward) to notify policemen. The code below shows the reaction on `PlayerInfo` tuples installed by the application.

```
def playerinfoTemplate := tuple: [PlayerInfo, ?uid, ?tid, ?location];
totam.whenever: playerinfoTemplate read: {
  GUI.displayPlayerPosition(tid, uid, location);
} outOfContext: {
  def matchingPlayerinfoTemplate := tuple: [PlayerInfo, uid, tid, ?loc];
  def tuple := totam.rdp(matchingPlayerinfoTemplate);
  if: (nil == tuple) then: { GUI.showOffline(uid) };
};
```

Whenever a `PlayerInfo` tuple is read, the player updates his GUI with the new location of that player. As `PlayerInfo` tuples are injected with a custom lease, they are automatically removed from the tuple space after their time interval elapses triggering the **`outOfContext:`** handler. In particular, opposing team members receive the player's location with a lease of 6 minutes, and policemen share their location with a lease of 1 minute. In the example, the **`outOfContext:`** handler grays out the GUI representation of a player if no other `PlayerInfo` tuple for that player is in the TOTAM system. If the `rdp` operation does not return a tuple, the player is considered to be offline as he did not transmit his coordinates for a while.

Note how the integration of context into reactions avoids having to write imperative code for inferring tuple perception. The underlying rule engine takes care of it, making the code easier to understand and maintain.

## 7.5 Formal Semantics

We now formalize our tuple space model by means of a calculus with operational semantics based on prior work in coordination [VC09, VO06]. This formalization is joint work with Christophe Scholliers. The syntax of our model is defined by the grammar shown in Table 7.3. $k$ identifies the type of the tuple: $+$ for a public tuple, $\oplus$ for a private tuple and $-$ for an antituple. A tuple $c$ is specified as a first order term $\tau$. $\tau^k_{x,t}\langle r\rangle$ indicates that the tuple with content $\tau$, type $k$ and time interval of its lease $t$, originates from a tuple space with identifier $x$ and is only perceivable when its context rule $r$ is satisfied. The context rule is considered optional and the notation $\tau^k_{x,t}$ should be read as $\tau^k_{x,t}\langle 1\rangle$, i.e. the context rule is always true. The antituple of a tuple $\tau^k_{x,t}$ is denoted by $\tau^-_{x,t}\langle 0\rangle$, i.e. its context rule is always false.

| | |
|---|---|
| $k ::= + \mid \oplus \mid -$ | Tuple Types |
| $c ::= \tau^k_{x,t}\langle r\rangle$ | Tuple |
| $S ::= \emptyset \mid c, S$ | Tuple Set |
| $P ::= \emptyset \mid A.P$ | Process |
| $C ::= \emptyset \mid (\llbracket S\rrbracket_x \mid C) \mid (P \mid C)$ | Configuration |
| $A ::= out(x, \tau, r, t) \mid inject(x, \tau, r, t) \mid rd(x, \nu) \mid in(x, \nu) \mid$<br>$outC(x, \nu) \mid whenRead(x, \nu, P_a, P_d).P \mid whenIn(x, \nu, P_a, P_d).P$ | Actions |

Table 7.3: TOTAM Tuples: Grammar

A process $P$ consists of a sequence of tuple space operations $A$. Tuples are stored in $S$ which is defined as a set of tuples composed by the operator (,). A tuple space with content $S$ and identifier $x$ is denoted by $\llbracket S\rrbracket_x$. A system configuration $C$ is modeled as a *set* of processes $P$ and collocated tuple spaces $\llbracket S\rrbracket_x$ composed by the operator $|$. An application consists of all $P \in C$.

Next to the grammar, we assume the existence of a matching function $\mu(\nu, \tau)$ that takes a template $\nu$ and a tuple content $\tau$, and returns $\theta$. $\theta$ is a substitution map of variable identifiers from the template $\nu$ to the actual values from $\tau$. A concrete value in this map can be accessed by $\theta_z$ that returns the actual value for $z$. The matched tuple can be accessed by $\theta_\tau$. We also assume the existence of a function *time* which returns a numeric comparable value indicating the current time. $r(S)$ indicates that the context rule $r$ is satisfied in the tuple set $S$.

The semantics of the our tuple space model is defined by the transition rules shown in Table 7.4. Every transition $C \xrightarrow{\lambda} C'$ indicates that a configuration $C$ can be transformed into a configuration $C'$ under the condition $\lambda$.

The $(OUT)$ rule states that when a process performs an `out` operation over a local tuple space $x$, the tuple is immediately inserted in $x$ as a private tuple with context rule $r$ and timeout $t'$. The process continuation $P$ is executed immediately. When a tuple is inserted in the tuple space $x$ with an `inject` operation as specified by $(INJ)$, the tuple is inserted in $x$ as a public tuple and is replicated to other tuple spaces as specified by $(RPL)$. This rule states that when a tuple space $y$ moves in communication range of a tuple space $x$, all tuples $\tau^k_{x,t}$ which are not private and are not already in $y$ will be replicated to $y$. The $(RD)$ rule states that to read a template $\nu$ from a tuple space $x$, $x$ has to contain a matching $\tau^k_{y,t}$ and the context rule of $\tau$ is satisfied in $S$. $(RD)$ blocks if one of these conditions is not satisfied. When $(RD)$ does apply, the continuation $P$ is invoked with substitution map $\theta$. Note that we do not disallow $x$ to be equal to $y$ in this rule. The (KILL) rule specifies that when both a tuple $\tau$ and its unique antituple $\tau^-$ are stored in the same tuple space, $\tau$ is removed immediately. The (TIM) rule specifies that when the timeout of a tuple $\tau$ elapses, its antituple $\tau^-$ is inserted in the tuple space.

The `in` operation is *guaranteed* to be atomically executed. In the semantics, it has

| | | | |
|---|---|---|---|
| $out(x, \tau, r, t).P \mid \llbracket S \rrbracket_x \mid C$ | $\xrightarrow{t'=time()+t}$ | $P \mid \llbracket \tau^{\oplus}_{x,t'} \langle r \rangle, S \rrbracket_x \mid C$ | (OUT) |
| $inject(x, \tau, r, t).P \mid \llbracket S \rrbracket_x \mid C$ | $\xrightarrow{t'=time()+t}$ | $P \mid \llbracket \tau^{+}_{x,t'} \langle r \rangle, S \rrbracket_x \mid C$ | (INJ) |
| $\llbracket \tau^{k}_{x,t} \langle r \rangle, S \rrbracket_x \mid \llbracket S' \rrbracket_y \mid C$ | $\xrightarrow[(x \neq y)]{\tau \notin S' \wedge (k \neq \oplus) \wedge}$ | $\llbracket \tau^{k}_{x,t} \langle r \rangle, S \rrbracket_x \mid \llbracket \tau^{k}_{x,t} \langle r \rangle, S' \rrbracket_y \mid C$ | (RPL) |
| $rd(x, \nu).P \mid \llbracket \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | $\xrightarrow[r(S)]{\mu(\nu,\tau)=\theta \wedge (k \neq -) \wedge}$ | $P\theta \mid \llbracket \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | (RD) |
| $\llbracket \tau^{-}_{y,t} \langle 0 \rangle, \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | $\xrightarrow{(k \neq -)}$ | $\llbracket \tau^{-}_{y,t} \langle 0 \rangle, S \rrbracket_x \mid C$ | (KILL) |
| $\llbracket \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | $\xrightarrow{t \leq time() \wedge (k \neq -)}$ | $\llbracket \tau^{-}_{y,t} \langle 0 \rangle, S \rrbracket_x \mid C$ | (TIM) |
| $in(x, \nu).P \mid \llbracket \tau^{k}_{x,t} \langle r \rangle, S \rrbracket_x \mid C$ | $\xrightarrow[(k \neq -)]{\mu(\nu,\tau)=\theta \wedge r(S) \wedge}$ | $P\theta \mid \llbracket \tau^{-}_{x,t} \langle 0 \rangle, S \rrbracket_x \mid C$ | (INL) |
| $in(x, \nu).P \mid \llbracket \tau^{+}_{y,t} \langle r \rangle, S \rrbracket_x \mid \llbracket \tau^{+}_{y,t} \langle r \rangle, S' \rrbracket_y \mid C$ | $\xrightarrow[(x \neq y)]{\mu(\nu,\tau)=\theta \wedge r(S) \wedge}$ | $P\theta \mid \llbracket S \rrbracket_x \mid \llbracket \tau^{-}_{y,t} \langle 0 \rangle, S' \rrbracket_y \mid C$ | (INR) |
| $outC(x, \tau).P \mid \llbracket \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | $\xrightarrow{!r(S)}$ | $P \mid \llbracket \tau^{k}_{y,t} \langle r \rangle, S \rrbracket_x \mid C$ | (OC) |
| $whenRead(x, \nu, P_a, P_d).P \mid \llbracket S \rrbracket_x \mid C$ | $\xrightarrow{1}$ | $rd(x, \nu).P_a.outC(x, \theta_\tau).P_d \mid P \mid \llbracket S \rrbracket_x \mid C$ | (WR) |
| $whenIn(x, \nu, P_a, P_d).P \mid \llbracket S \rrbracket_x \mid C$ | $\xrightarrow{1}$ | $in(x, \nu).P_a.outC(x, \theta_\tau).P_d \mid P \mid \llbracket S \rrbracket_x \mid C$ | (WI) |

Table 7.4: Operational Semantics

been split into a local rule $(INL)$ and a remote rule $(INR)$. $(INL)$ works similarly to $(RD)$, but it removes the tuple $\tau^k_{x,t}$ originated by the local tuple space $x$ and inserts its antituple $\tau^-_{x,t}$. $(INR)$ states that when the `in` operation is matched with a tuple published by another tuple space $y$, $y$ must be one of the collocated tuple spaces (i.e. be in the configuration). Analogously to $(INL)$, the tuple is removed and its antituple is inserted. The $(OC)$ rule states that to move out of context a tuple $\tau$ from a local tuple space $x$, $x$ has to contain $\tau$ (possibly its antituple) and its context rule is *not* satisfied. The $WR$ rule states that a `whenRead` operation performed on the local tuple space $x$ with template $\nu$ and processes $P_a$ and $P_d$, is immediately translated into a new parallel process and the continuation $P$ will be executed. The newly spawned parallel process is specified in terms of performing a `rd` operation followed by an `outC` operation. A `rd` operation blocks until there is a tuple matching $\nu$ in the local tuple space. The continuation $P_a$ is then executed to subsequently perform an `outC` which blocks until the tuple is no longer perceivable. Finally, the continuation $P_d$ is invoked whereafter the process dies. The $WI$ is specified analogously but as it models a `whenIn` operation, it performs a `in` operation rather than a `rd` one. The `wheneverRead` and `wheneverIn` operations have been omitted as they are trivial recursive extensions of `whenRead` and `whenIn`, respectively.

Note that $(KILL)$ does not remove antituples. This has been omitted to keep the semantics simple and concise. By means of $(RPL)$, the antituple of a tuple $\tau$ is only replicated to those systems that received $\tau$. In our concrete implementation if a system cannot be reached, the removal of the antituple is delayed until the timeout of its tuple elapses (which inserts an antituple as specified by $(TIM)$). An antituple can only be removed once there are no processes in the configuration which registered an `outC` operation on the original tuple.

## 7.6 Implementation Status

As previously mentioned, TOTAM has been implemented in AmbientTalk. In order to use the TOTAM middleware, developers need to import the TOTAM module accessible via the `lobby.at.lang.totam` namespace. This module provides the core model including the scoping mechanism based on propagation protocol and the leasing model. The context rules that can be attached to a tuple have been implemented as a separated module extending TOTAM that is accessible via the `lobby.frameworks.tuples` namespace. The rule engine used in its implementation incorporates a truth maintenance system built on top of a RETE network [For89]. A RETE network optimizes the matching phase of the inference engine providing an efficient derivation of context rule activation and deactivation. The network has also been optimized to allow constant time deletions by applying a *scaffolding* technique [Per90]. For details about the engine and its performance, we refer to [SP07].

A TOTAM system relies on AmbientTalk's service discovery facilities (based on multicast using UDP) to discover other systems in the network. Flikken's initial experimental setup was a set of HTC P3650 Touch Cruises phones running on J2ME (CDC) and communicating by means of TCP broadcasting on a wireless ad hoc WiFi network. It remains future work to port the current Java AWT GUI to the Android platform.

We have written other AmbientTalk applications which use TOTAM and run on Android devices. An example is the Guanotes application used to explain TOTAM's programming API in Section 7.3. Further details about it can be found in [GBLCS$^+$11]. TOTAM has also been used to prototype a mobile bulletin application in collaboration with the Brussels public transport company (STIB/MIVB). The application works like the bulletin boards that one sometimes sees at the exit of supermarkets with messages such as "Im looking for a housekeeper", but integrated in a small part of the Brussels public transportation system (3 buses)[7]. Passengers can use an Android device to post messages on the bus and read messages that were posted by previous passengers. Crossing buses exchange messages such that they get percolated through the transportation network.

## 7.7 Discussion

In Section 6.1.2, we postulated four criteria to which a good leasing model for MANETs should adhere. In this section, we discuss how TOTAM supports ambient-oriented leasing by evaluating it in the light of these criteria.

**Criterion#1: Leasing an Intermittent Connection** TOTAM combines leasing with a non-blocking communication model based on tuple spaces. In order to deal with the intermittent connectivity inherent to MANETs and to increase data availability, TOTAM replicates tuples to other tuple spaces in the network. However, tuples carry a propagation protocol which allows them to limit transportation when necessary. By default, applications are not aware of the intermittent disconnections of other TOTAM systems in the network since the model abstracts the configuration of the network. When a higher degree of context-awareness is required, tuples can contain context rules describing the runtime conditions under which the tuples should be visible in the receiving tuple space. In order to deal with permanent disconnections, programmers can inject tuples with a *lease* which determines how long the tuple should remain in the tuple space.

Since leases are encoded as part of a tuple's propagation protocol, and tuples are replication among TOTAM systems, this may introduce issues of clock synchronization. In particular, the system cannot provide strong guarantees about the expiration

[7] A demo of the mobile bulletin application deployed on buses at MIVB is available at `http://www.youtube.com/watch?v=N7mxaPftod4`

time. In other words, it could be that some copy of a tuple is still unrightfully active in the system, causing the tuple to be perceivable by applications. At worst, the asynchrony causes a tuple to be subject to **read** operation. However, the tuple will not be available for in operations because for a in operation to succeed the owner of the tuple has to be contacted. When contacting the owner, the requesting tuple space will be informed that the lease's tuple is expired.

To sum up, the combination of leasing with a replication-based tuple space model provides a coherent tuple space-based abstraction that deals with both intermittent and persistent failures.

**Criterion#2: Leasing Management Patterns** TOTAM supports leases based on fixed time-based terms. Developers can determine the time interval of a lease attached to a tuple by means of the **inject:** operation. While renewal of such leases is not supported, revocation is possible by explicitly retracting the tuple from the network (calling the **retract** method on the subscription object of the **inject:** operation). As explained before, this results in the injection in the TOTAM network of an antituple for the tuple.

Currently TOTAM does not offer any built-in leasing variants similar to the ones we introduced for our ambient-oriented leasing for a distributed object-oriented model (presented in the previous chapter). Nevertheless, they can be built by means of custom propagation protocols (as we discuss below).

**Criterion #3: A Customizable Leasing Framework** Propagation protocols provide developers with a framework where to construct custom leasing variants. In constrast to the open implementation of leased references, propagation protocols have not been designed especially for supporting the development of leasing variants. Rather, their main goal is to provide fine-grained control over the propagation of tuples in the network. Nevertheless, our implementation features a leasing propagation protocol prototype object, which can be extended to build other leased tuple kinds.

It is important to notice that propagation protocols can be developed separately from tuple content. As such, once developers built their custom leasing variants in a propagation protocol, the protocol can be applied to different kinds of tuples, promoting its reusability across different applications.

**Criterion#4: Symmetric Expiration Handling** Expiration handling is supported in TOTAM by registering **outOfContext:** listeners on reactions (cf. Table 7.1). When the lease of the matching tuple expires, all registered handlers for that tuple will be triggered allowing programmers to apply compensating actions, e.g., inject a new tuple or cancel the injection of another tuple. Note that in the case of registering a reaction for an in operation, only the process which actually removed the tuple is notified. These listeners allows processes "interested" in a tuple to be notified. However, the process injecting a tuple cannot react to its expiration because TOTAM does not offer dedicated listeners to react to the expiration of tuples matching a template (even if they were never read or removed from the tuple space). This forces processes injecting the tuple to also register a reaction for it, even if they are not interested in its content (since they already have this information).

## 7.8 Limitations and Future Work

In this section, we highlight aspects of TOTAM which can be improved on, and give some directions for future work.

**Leasing Model** To the best of our knowledge, TOTAM is the only replication-based tuple space model targeting a mobile environment that includes a leasing model. The leasing model is, however, a restricted variant of ambient-oriented leasing that we have proposed in the previous chapter. It lacks expressive support for leasing management patterns and provides restricted symmetric expiration handling. Nevertheless, we believe that TOTAM provides a solid grounding towards *leased-based coordination* in MANETs. Further research is required at least in two areas.

First, which leasing patterns are useful in a data-driven model? In tuple spaces, a tuple represents a self-sufficient piece of information conveyed by the distributed system. In addition, in a replication-based model like TOTAM, different copies of the same tuple can coexist in the system. This means that the tuple does not have an identity as such. This may not be a problem since tuples are anonymous, and meant to be consumed by one process. But it raises fundamental questions about which lease management operations make sense. For example, one could imagine that it is useful to let the system implicitly renew the lease of certain types of tuples if they are highly demanded, e.g., the tuple can potentially match several registered reactions. However, it is very difficult to determine the guarantees provided for such a leasing variant since maintaing all copies in synchronization in such a dynamic network topology is impractical.

Second, how can programmers expressively deal with expired leases? From our experience with TOTAM, we believe it may be useful to provide dedicated expiration listeners which will allow the application to apply compensating actions even if the tuple was not read or remove. For example, in the context of Flikken, it will be useful to be able to replace a `PlayerInfo` tuple with a new one when it expires. Currently, this has to be manually encoded by injecting tuples at regular intervals.

**Tuple Space Descriptors** Tuple space descriptors are exchanged between two locations when they meet for the first time and whenever a location decides to change its description. In case descriptors stay constant and prevent the propagation of tuples, they can drastically reduce the burden on the network. In the other case when descriptors change a lot or do not prevent the transportation of tuples, the danger exists that the network traffic gets dominated by the transmission of descriptors.

In [SGD09], we evaluate the use of tuple space descriptors in terms of network traffic. In the worst case, a tuple needs to be transported to all connected locations (i.e., the tuple is not scoped and floods the network) and all connected systems change their descriptors for every transmitted tuple. The overhead of exchanging the descriptors will be quadratic to the number of connected locations over time. In the best (meaningful) case, the sent tuples are sculpted to be sent only to one system and the tuple space descriptors do not change over time, e.g., the tuple hops from system to system in a ring. The overhead of exchanging the descriptors will be equal to the size of the ring. All in all, the use of tuple space descriptors in combination with scoped tuples has the potential to drastically reduce the network traffic when 1) the tuples are prevented from hopping around and 2) the descriptors do not change often relative to the number of tuples in the system. However, how often do the descriptors change is highly application-dependent. For example, in Flikken, tuple descriptors do not change over time because players were not allowed to switch teams. In Guanotes, the tuple descriptor changes when the end-user updates its user profile. As future work, we would like to conduct experiments with devices to measure the costs of changing tuple space descriptors for different applications as well as for different propagation strategies.

## 7.9 Notes on Related Work

Before concluding this chapter, we discuss related work with regard to the various concepts integrated in TOTAM, namely its leasing model, scoping mechanism and support for context-awareness.

**Leasing Models** Typically, a tuple space stores tuples which may never be subject to a remove operation, and which may never be garbage collected. To solve this problem, some traditional tuple space approaches such as Objective Linda [Kie96], JavaSpaces [FAH99] and TSpaces [WMLF98], augmented Linda's operations with a timeout: if a matching tuple is not found within the timeout, the operation returns an error. In JavaSpaces and TSpaces, a tuple can be inserted in the tuple with a *lease time* denoting the maximum amount of time before the tuple is automatically removed from the tuple space. However, the centralized nature of those approaches does not make them applicable in a mobile environment.

To the best of our knowledge, Tiamat [ME03] is the only tuple space model for the mobile environment that includes a leasing model. Tiamat follows a federated tuple space model in which each operation is leased. Interestingly, the authors describe that in Tiamat "leases may be based on time or on other measures such as the number of remote instances contacted". Unfortunately, no code examples are provided to see how programmers can declare leases based on such conditions. Tiamat leasing model incorporates the concept of expiration in the tuple space operations, but it does not provide listeners which allows application to react to them.

In [MZ09], Mamei et Zambonelli remark the importance of incorporating a garbage collection mechanism to TOTA to remove unused tuples. In TOTA, one can encode a leased tuple such as the one described in our work by means of a custom propagation rule. In particular, the propagation rule needs to update its content to take into account the notion of time, and stop its propagation. In fact, as explained before in Section 3.3.2.2, the `MessageTuple` class described in [MZ09] defines a tuple that floods the network and deletes itself after some time has passed. In TOTAM, a lease is orthogonal to the propagation protocol, so programmers do need to manually encode in the propagation rule the passage of time, nor the retraction of the tuple from the TOTA network upon a `delete` operation.

In [ORV05], Ommici et al. extend ReSpecT [DNO98], logic-based tuple space language with the notion of time. ReSpecT tuple centers behave like tuple spaces whose behaviour is specified in terms of reactions to events occurring in the tuple space. The notion of time is introduced into a tuple center with (1) some temporal predicates to get information about tuple center and event time, and (2) timed reactions which specify reactions triggered by time events. Based on this extension, they discuss how some abstractions that could be built by introducing leasing into tuples similar to what TOTAM supports. An interesting topic of future work would be to investigate whether introducing leasing in TOTAM along the principles of (timed) tuple centers will allow us to build time-based coordination patterns in a more modular way than our current solution based on propagation protocols.

**Scoping Mechanisms** A number of approaches support scoping mechanisms in the context of tuple spaces. Coordination with Scopes [MW00] introduces the concepts of scope for a tuple space. A scope represents a view on a flat tuple space. A set of operations is defined on those views allowing scopes to be joined, nested, intersected and subtracted. Tuples may thus be visible from several different scopes. This mechanism is mainly used to structure tuple spaces according to different viewpoints on a flat tuple space. However, scopes do not limit the propagation of tuples, i.e., tuples are propagated to other tuple spaces but may not be visible for certain scopes. Since the system was not devised for mobile computing applications, they rely on a centralized infrastructure. In contrast, TOTAM does not rely on any fixed infrastructure and tuples can be propagated through spontaneously formed MANETs.

CAMA [IA06] is an agent-based tuple space system which allows the definition of a scope which agents can join and leave. Scopes are defined as containers in a tuple space and can be nested in order to form hierarchical structures. This notion of scope improves on coordination with scopes since inserted tuples are only transmitted to agents which reside in the same scope. However, in order to send tuples to other scopes the agent first needs to change its scope. Tuples which are inserted in a specific scope can not be propagated automatically to other scopes. In TOTAM, by allowing the tuple itself to decide whether it should be propagated, more fine-grained sharing strategies can be expressed.

L2imbo [DFWB98] is a tuple-space based platform for mobile computing which provides special features for quality of service. Similar to CAMA, L2imbo introduces the concept of multiple tuple spaces but suffers from the same limitations as tuples do not have the ability to decide to which tuple space they should be propagated. It is interesting to note that L2imbo supports time-outs associated with tuples. This makes possible for the system to reorder tuples to make optimal use of the available network connectivity. Similar to CAMA, L2imbo introduces the concept of multiple tuple spaces but suffers from the same limitations as tuples do not have the ability to change to which tuple space they should be propagated.

Evolving tuples [SJ07] have a field destination that they can change while they are hopping. This destination field is used to determine where the tuple will be transmitted to after leaving a host. However, the destination field can only be a broadcast address or a specific host address. In order to send the tuple to two broadcast addresses the programmer will have to read the tuple and reinsert it to another broadcast address. Our approach uses semantic information to determine where it can be transmitted thus allowing more fine-grained propagation rules.

Publish/subscribe systems are closely related to tuple spaces as they provide similar decoupling properties [EFGA03]. Some of those systems have explored the concept of scope for events. In scoped REBECA [LMMB02] systems can create a scope in which events will be published. A scope can be extended and therefore form a tree of scopes. Subscribers will only receive events of publishers which are in the same scope or have a common ancestor in the scope hierarchy. Similar to CAMA, publishing an event in an other scope requires the publisher to change it scope first. STEAM [MC03] allows publish/subscribe based on physically location, but it is hard to describe scopes based on semantic information as shown in the ambient game. Location-based publish/subscribe [EGH05] suffers from the same limitation.

**Context-Awareness.** Finally, we discuss related systems modeled for context-awareness and show how context-aware tuples differ from them. In TOTA, tuples themselves decide how to replicate from node to node in the network. Because tuples can execute code when they arrive at a node, they can be exploited to achieve context-awareness in an adaptive way. However, programming such tuples has proven to be difficult [MZ04]. TOTA, therefore, provides several basic tuple propagation strategies. None of these propagation strategies addresses the tuple perception problems tackled by our approach. Writing context-aware tuples in TOTA would require a considerable programming effort to react on the presence of an arbitrary combination of tuples as it only allows reactions on a single tuple.

GeoLinda [PCBB07] augments federated tuple spaces with a geometrical read operation `read(s,p)`. Every tuple has an associated shape and the `rd` operation only matches those tuples whose shape intersects the addressing shape `s`. GeoLinda has been designed to overcome the shortcomings of federated tuple spaces for a small subset of potential context information, namely the physical location of devices. As such, it does not offer a general solution for context-aware applications. In contrast, we propose a general solution based on context rules, which allows programmers to write application-specific rules for their tuples. Moreover, in GeoLinda the collocation of devices still plays a central role for tuple perception which can lead to erroneous context perception.

EgoSpaces provides the concept of a *view*, a declarative specification of a subset of the ambient tuples which should be perceived. Such views are defined by the *receiver* of tuples while in context-aware tuples it is the other way around. Context-aware tuples allow the *sender* of a tuple to attach a context rule dictating the system in which state the receiver should be in order to perceive the tuple. EgoSpaces suffers from the same limitations as federated tuple spaces since, at any given time, the available data depends on connectivity [JR04].

In publish/subscribe models, context-aware publish subscribe (CAPS) [FR07] is the closest work as it allows certain events to be filtered depending on the context of the receiver. More concretely, the publisher can associate an event with a *context of relevance*. However, CAPS is significantly different from context-aware tuples. First, CAPS does not allow reactions on the removal of events, i.e., there is no dedicated operation to react when an event moves out of context. Moreover, it does not provide coordination of distributed parties, i.e., atomic removal of events is not supported. And last, the context of relevance is always associated to a physical space.

## 7.10 Conclusion

In this chapter, we have presented a novel tuple space model for MANETs called TOTAM. TOTAM combines ideas from both federated and replication-based tuple spaces into a consistent tuple space-based framework that is designed around the principles of *ambient-oriented leasing*. The design and implementation of TOTAM also demonstrates that the principles of ambient-oriented leasing are independent of the particular communication paradigm used to incarnate non-blocking communication, and are also applicable to a data-driven programming paradigm such as tuple spaces.

We conclude this chapter by stating the novelty of TOTAM:

- It introduces a programming style under the form of *context rules* to support the development of context-aware MANET applications. Unlike existing tuple space approaches, *only* the subset of tuples which should be perceivable, is made accessible to applications in TOTAM. This is achieved by the use of context rules combined with the introduction of a rule engine in the tuple space system which takes care of inferring when a context rule is satisfied.

- It extends the TOTA-like replication-based model with the use of tuple space descriptors to determine the scope of tuples *before* they are being transmitted. This enhances privacy and decreases the burden on the network traffic in a wide range of applications.

- It integrates the concept of *leasing* into tuples. This allows developers to determine upper boundaries on the availability of tuples in the system.

- It introduces the concept of *antituples* into a replication-based model to enable the "unsending" tuples injected to the network. Antituples are injected into the TOTAM network upon removal or retraction operations, avoiding programmers to manually encode them in terms of propagation protocols.

# Chapter 8

# Evaluating Ambient-Oriented Leasing

We conclude the programming language design part of this dissertation by evaluating the novel programming abstractions for ambient-oriented leasing in the light of the criteria presented in Section 2.4. We evaluate language support for ambient-oriented leasing for both a distributed object-oriented model (presented in Chapter 6) and a distributed tuple space-based model (presented in Chapter 7).

In this chapter, we employ the AmbientTalk language's full names (AmbientTalk/2 and AmbientTalk/M) to clarify our discussion.

## 8.1 Evaluation w.r.t. the Criteria for a Failure Handling Model for MANETs

Recall from Table 2.1 that we identified eight criteria for a failure handling model to be used in a MANET. We now describe how AmbientTalk, when extended by support for ambient-oriented leasing, conforms to these criteria. For each criterion, we first recapitulate its statement and then discuss the extent to which it has been achieved by the two incarnations.

**C1 Decoupled Communication** *allows processes to deal with the effects of intermittent disconnections by enabling them (1) to communicate while being disconnected (decoupling in time), (2) to interact anonymously without knowing their exact addresses (decoupling in space), (3) to remain responsive to communication (decoupling in synchronization) and (4) to abstract from the concrete number of processes communicating with (arity decoupling).*

In AmbientTalk/M, remote object references by default already decouple client and service objects in time. This is witnessed by the asynchronous message protocol in the transmitter-receptor model (summarized in Table 5.1). The transmitter of a remote reference behaves as the outbox of an actor, in which the client object can schedule messages to be transmitted to the corresponding service object. Such time-decoupling characteristic is combined with a lease term in the case of leased references. The processing of a message send can then be bounded by means of the `@Due` annotation. In TOTAM, time-decoupled communication is supported as well because the tuple space adheres to a replication-based model in which tuples are replicated among collocated devices.

AmbientTalk/M inherits from AmbientTalk/2 a limited form of space decoupling by means of its publish/subscribe service discovery mechanism (cf. Section 4.4.2). It allows objects to discover each other in the network by means of type tags, i.e., without knowing their exact address. As we mentioned in Section 6.9, leased references do not provide any built-in space decoupling mechanisms. In this regard, leased references are no improvement over the default far references implemented by the transmitter-receptor model. Instead, we have exploited support for space decoupling in our exploration of ambient-oriented leasing in tuple spaces. As our survey on related work shows, this is a characteristic that is naturally supported in tuple spaces.

AmbientTalk/M's execution model is totally sculpted for decoupling client and service objects in synchronization. This is also a property that AmbientTalk/M inherits from the AmbientTalk/2 actor-based concurrent model. As a result, any abstraction built on top of the kernel language maintains such characteristic. It includes leased references and TOTAM. This explains why TOTAM does not feature any of the traditional blocking operations from tuple spaces. Rather, its programming API is entirely built around the concept of reactions.

Finally, although remote object references (and leased references) are by default point-to-point communication links, AmbientTalk/M supports arity decoupling in the underlying transmitter-receptor model by means of its reference marshalling protocol (summarized in Table 5.2). Arity decoupling is endorsed by TOTAM. Recall that all communication in a tuple space model happens in a one-to-many fashion. However, in contrast to other replication-based tuple space models, TOTAM guarantees atomicity for removal operations. This supports synchronization of communicating parties since a tuple can only be removed by *one* process.

**C2 High-level Representation of Failures** *enables processes to determine when their logical connections or exchanges of data have terminated.*

In this work, we have studied the concept of leasing as a programming language abstraction that provides a high-level representation for failures. AmbientTalk/M provides the concept of a lease as a first-class object that exposes a number of methods that can be overridden to specialize the behaviour of leasing. In contrast to prior leasing models, AmbientTalk/M's language support provides developers with a number of fine-grained scoping abstractions to specify which lease should be used when and where. First, a leased reference offers leasing on a per-object basis. One of the key features of leased references is that it decouples the concept of a failure in the *logical connection* between client and service objects, from a failure in the underlying *physical connection*. Second, `@Due` annotation can be used to specify leasing semantics for individual message sends, limiting the lifetime of a message sent via a leased reference. Finally, leasing strategies and leased message protocols enable developers to apply the same leasing semantics to an entire group of objects or messages.

The use of leasing in the TOTAM tuple space model allows developers to define an upper bound on the lifetime of a tuple in the tuple space. By integrating leasing into tuples, programmers can actually distinguish *valid tuples* from *stale tuples* independently from the composition of the TOTAM network (i.e., the connectivity of the TOTAM tuple spaces participating in the underlying network). As such, TOTAM provides leasing on a per-tuple basis.

**C3 Reacting to Network Connectivity** *enables processes to monitor network connectivity of other processes and react if necessary.*

As described in Section 4.4.1, AmbientTalk/2 already provides two event handlers which can be registered to monitor the states of a far reference. The `whenever: disconnected:` and `whenever:reconnected:` observers are triggered whenever a far reference becomes disconnected or reconnected. AmbientTalk/M inherits them but offers them on leased references as well.

TOTAM, on the other hand, does not feature linguistic support for reacting to network connectivity of the underlying TOTAM network. This needs to be encoded manually by injecting leased tuples which define the presence of a device in the network. We will further discuss this limitation in Chapter 12.

**C4 Local Failure Recovery** *enables processes to recover from failures based on their local state as much as possible.*

Leased references support local failure recovery by providing buffering of messages while the reference is disconnected. TOTAM, on the other hand, naturally supports local failure recovery because it employs a replication-based tuple space model in which intermediary nodes are used to increase the availability of tuples in the face of partial failures.

Recall from our discussion about related work that in TOTA, tuples carry a propagation rule which enables programmers to scope the tuple dissemination process. TOTAM extends such scoping support with the use of tuple space descriptors to determine the scope of tuples *before* they are actually transmitted.

**C5 Application-dependent Failure Handling Strategies** *enables processes to select the most appropriate strategy to react to a partial failure.*

AmbientTalk/M provides `when:expired:` handlers to enable programs to react to the expiration of a leased reference. Since these handlers can be installed on either side of a leased reference, both client and service objects can gracefully deal with the termination of their logical connection and schedule appropriate compensating actions.

TOTAM enables a limited form of application-dependent failure handling strategies by registering `outOfContext:` handlers on reactions. As previously mentioned, when a leased tuple expires, all registered handlers for that tuple will be triggered allowing programmers to apply compensating actions, e.g., inject a new tuple or cancel the injection of another tuple.

Note that AmbientTalk/M provides support to build custom failure handling strategies to react to network disconnections by means of the environmental context protocol provided in the transmitter-receptor model (summarized in Table 5.2.2.3).

**C6 Relaxing Soundness** *enables connections or exchanges of data to remain valid in the presence of intermittent disconnections.*

In our object-oriented incarnation of ambient-oriented leasing, leased references relax soundness because a network disconnection does not cause the leased reference to block (as happens in RPC models employed such as in Java RMI) or break (as happens in non-blocking models not designed for MANETs such as in E). Since leased references by default buffer messages sent while they are disconnected, computation is not lost in the presence of intermittent disconnections.

In short, we observe that leasing in combination with disconnection operation techniques (such as buffering) are the key enabler to relax soundness on the memory management scheme of a distributed object model for MANETs. In a tuple space model, on the other hand, relaxing soundness is achieved by employing a replication-based model. This was one of the reasons why we chose to extend TOTA's tuple space model for our second incarnation of ambient-oriented leasing.

**C7 Contractual Memory Management** *enables processes to reclaim unused resources in the presence of permanent disconnections by agreeing on the lifetime of data before it is actually shared.*

Both incarnations of ambient-oriented leasing naturally support contractual memory management by embracing the concept of leasing. Once all leased references to a service object have expired, the object becomes subject to garbage collection as soon as it is no longer locally referenced. Since the duration of the lease is agreed beforehand, if client or service objects are subject to a partial failure, the language remains in control of the garbage collection of both ends of the reference when the lease expires.

In TOTAM, contractual memory management is supported by combining leases with the notion of antituples. As a result, either a tuple can be reclaimed when its lease expires, or it can be opportunistically reclaimed upon retraction or removal (which implicitly injects an antituple in the TOTAM network that traces the copies of the matching tuple and removes them). The linguistic support provided by leased tuples and antituples precludes programmers from having to manually encode tuple lifetime management in terms of propagation protocols as it is the case in TOTA.

**C8 Forcing Failures** *enables processes to trigger failure handling code even if no physical network failure ocurred. This criterion plays an important role for tool support as it enables tools like debuggers or unit testing tools to reproduce a number of network configurations and expose the application to them before its deployment.*

AmbientTalk/M allows applications to cause partial failures of objects. Recall from Section 5.3.2, that the `takeOffline:` construct allows one to *permanently* disconnect an object, while the `disconnect:` constructs allows one to *temporarily* disconnect an object (which can be later reconnected by calling the `reconnect` method). We initially introduced these constructs as primitive operations in AmbientTalk/2. However, thanks to the new reflective architecture of AmbientTalk/M, now they can be provided reflectively, reducing the kernel language size. In particular, the reference management protocol (cf. Table 5.3.1.1) makes this possible as it reifies the actor's object reference table. The aforementioned constructs can be directly used with leased references to disconnect both sides of the reference.

TOTAM supports forcing failures by means of a dedicated operation, i.e., `goOnline`. Recall from Table 6.1 that this operation causes the tuple space become to available in the TOTAM network, and returns an object whose `cancel` method can be used to disconnect the tuple space. The underlying implementation makes use of the `disconnect:` construct to disconnect the tuple space object from the network.

## 8.2 Concluding Remarks

Our analysis of our proposal shows that AmbientTalk/M already provides some forms of decoupled communication (in particular, decoupling in time, and synchronization (**C1**)), support for reacting to network connectivity (**C3**) and support for forcing failures (**C8**). Leased references complement such forms of decoupled communication with a high-level representation of failures (**C2**). Such integration of leasing into remote object references also supports local failure recovery (**C4**) (by buffering messages) and application-dependent failure handling strategies (**C5**) (by means of `when:expired:` handlers).

AmbientTalk/M's lack of communication abstractions which are space and arity decoupled is covered by TOTAM. TOTAM combines a tuple-space style of decoupled communication with a high-level representation of failures (**C2**) based on leasing as well. Such integration of leasing into tuples also supports local failure recovery (**C4**) (by replicating tuples among tuple spaces in the network) and application-dependent failure handling strategies (**C5**) (by means of `outOfContext:` handlers).

Finally, both leased references and TOTAM relax soundness (**C6**) and contractual memory management (**C7**) as a natural result of integrating leasing with a time-decoupled communication model.

Together AmbientTalk/M, leased references, and TOTAM, are able to directly address the eight criteria for adequately supporting failure handling in MANETs. Moreover, they provide this failure handling in an object-oriented way as well as in a data-driven way (in tuple spaces).

We postpone a discussion of the overall technical and conceptual limitations of our proposal until Chapter 12.

# Part II

# Partial Failures in Software Development Tools

# Chapter 9

# Related Work

Software tools are an integral part of application development since they help developers better face the complexities of software development. Debuggers contribute to this task by allowing developers to trace back the execution of a program, looking for the places in which it deviates from the intended behaviour. Next to assist in the arduous task of finding errors, debuggers also improve program comprehension as they help developers get a better understanding of the dynamic behaviour of a program. In this second part, we explore software development tools for MANET applications in the form of a debugger. Developing such applications is hard due to the highly changing nature of MANETs. Thus, developing software tools, specifically debugging tools, is more necessary than ever.

We start this second part of this dissertation by surveying the state of the art of debugging techniques and tools for concurrent and distributed systems. We mainly focus our discussion on approaches that work on actual program execution, rather than dynamic or static analysis approaches which use source code to check certain properties, or observe testing scenarios. In the next chapter, we will discuss their shortcomings and introduce a set of requirements to build an ambient-oriented debugger (cf. Section 10.1). Subsequently, we describe the design of an ambient-oriented debugger in Section 10.2, and a concrete instantiation for AmbientTalk programs in Section 10.3. Chapter 11 concludes this part by describing a user study for REME-D, the results of which show that developers actually see the added value of REME-D for making ambient-oriented programming in AmbientTalk easier.

## 9.1 Distributed Debugging Support

A lot of research has been conducted in developing debugging tools and techniques for concurrent and distributed systems, resulting in a large number of tools. In 1993, Pancake and Netzer published one of the most relevant bibliographic efforts in the field including 293 entries [PN93]. This effort persisted in their online bibliography [PN04] which, on the last update in 1997, counted no less than 659 entries about technical reports, journal and conference papers, and Phd dissertations dealing with parallel and distributed debuggers. However, many of those techniques are now outdated because of the rapid advances in the latests years in both hardware and software [WCS02], e.g., GUI-based frontends for debuggers is nowadays a given.

In this section, we present a brief survey of the state of the art of distributed debuggers. Based on the McDowell et al. survey on debugging concurrent programs [MH89], we categorize distributed debuggers into two main families: event-based debuggers (also known as *log-based* debuggers) and traditional breakpoint-based debuggers. The first approach consists in inserting log statements in the program to be able to generate a trace log (also called *event history*) during its execution. By browsing the event history, developers can examine the behaviour of the program execution. Since most event-based debuggers only support browsing after program completion, these tools are often referred to as *post-mortem* debuggers. Breakpoint-based debuggers, on the other hand, execute the program in *debug* mode under the control of a dedicated debugger that allows programers to pause/resume program execution at certain points, inspect program state, and perform step-by-step execution. As such, breakpoint-based debuggers are typically described as *online* or *interactive* debuggers.

### 9.1.1 Breakpoint-based Debuggers

Most distributed breakpoint-based debuggers are an extension of a conventional sequential debugger in which processes are considered to be the basic building blocks, and communication between processes happens by means of message passing. The debugger consists of a collection of sequential debuggers (e.g., gdb and dbx) acting as the *back end* of the debugger, each controlling the execution of a concurrent or remote process. A centralized GUI or console then controls and coordinate the activities of such dedicated sequential debuggers, acting as the *front end* of the debugger. Most-well known examples of these types of debuggers include research prototypes like p2d2 [Hoo96], Node Prism [SAB$^+$94], Net-Dbx [NNE04], and CDB [WCS02], and commercial debuggers such as TotalView [Got09], IBM's Distributed Debugger [MMP$^+$96], and Allinea DDT [DDT12].

Breakpoint-based debuggers aim to bring the well-known toolbox of sequential debugging to a distributed setting. They typically offer the traditional commands to stop, inspect program state, and step-by-step execution of a running program. Stack traces, for example, give the developer an idea of what has happened before during the execution of the program, answering the question of how the developer got to the current point in the execution. Despite the fact that the stack view does not show total causality, in most cases tracing back the stack is enough to find the cause of a bug [SCM09]. When this does not uncover the cause, breakpoints make it easier to mark interesting places in the execution. Some of the breakpoint-based debuggers allow to set breakpoints on statements of one process (e.g., TotalView) or a set of processes (e.g., p2d2, Node Prism). An interesting alternative to traditional breakpoints is *message breakpoints* [Wis97]. A message breakpoint stops all receiver processes of the next message sent by a process. The combination of a message breakpoint with a traditional breakpoint on the send statements provides a single step execution which can be used to monitor message sending and follow the execution of a distributed program.

One of the main critiques to breakpoint-based debuggers is that the level of granularity of commands is limited and too-fine grained as the focus is on the source code, making debugging complicated and the amount of information overwhelming [TP05]. Millipede aims to solve this by introducing a multi-level debugging approach which consists of three levels: the sequential level (controlling the intra-process execution), the message level (controlling messages interchanged between processes), and finally a protocol level (concerned with communication protocol). At a sequential level, it combines interactive features (allowing developers to attach gdb to a running process) while logging debugging information in a SQL database for future inspection. At a message level, Millipede logs messages sent and received by a process, and allows to stop and step the execution of one PVM API call. At protocol level, Millipede allows developers to specify rules over message sends or receipts specifying the expected program behaviour which is checked for correctness when the debugger logs information about message sends and receipts.

For the most part, all these distributed debuggers assume a stable network infrastructure. Fragile communication channels are assumed to be handled at the application level, i.e., communication failures are seen as an application-level errors. However, in a mobile setting, it is desirable that the debugger gracefully deal with network disconnections. A relevant exception is TotalView which supports open debugging sessions to some extent by relying on the underlying MPI middleware to manage and connect to new or independently started processes. Developers can dynamically attach processes running on nodes that execute a TotalView debugging agent to a debugging session. This gives a degree of freedom in the configuration of a debugging session, making it attractive for ambient-oriented applications. However, the target application still needs to be compiled in a special way in order to be able to interact with TotalView's debugging agent before it can be dynamically included in the debugging session.

### 9.1.2 Event-driven Debugging Tools

A large body of distributed debugging techniques follow an event-based approach. Event-based debuggers [MH89] conceive the execution of a program as a sequence of "events" whose definition differs. For example, an event may be a MPI API call, read/write memory, send/receive functions, etc. The debugger records the history of the events generated by the application, which can then be used to either browse the events once the application is finished [SCM09, FPK$^+$07], or to replay the execution of a program in order to recreate the conditions under which the bug was observed [TKV02, NM92, Els89, LMC87]. How to analyze the history of events also varies from presenting directly the raw data to the user for inspection, to relying on graph-based analysis methods, or supporting graphical visualization techniques (e.g., time-space diagrams [FHL98, MCC$^+$95], message and process order views [SCM09]).

Event-based debuggers have been criticized mainly because the recording process is costly (due to the overhead of collecting and saving information) and browsing an event history does not scale since manually inspecting huge traces becomes cumbersome and difficult [MH89]. Many research efforts have focused on reducing the amount of events recorded or presented to the user [PTP07, NM92, RK98b, Els89]. For example, Amoeba [Els89] supports customizable filters (also called *global filters*) that remove events that are of no interest to the user. Amoeba performs post-processing of the resulting events using *recognizers* (state machines acting for patterns of events). Only the events that also pass such local filters are then presented in Amoeba's user interface.

Event-based debuggers fit well with a non-blocking concurrency model as message sends and receipts can be represented as separate events. A partial order of such events would accurately reflect the behaviour of a distributed application. Some approaches explore a partial order of the event history based on the *happened before* relation for browsing [SCM09] or replay [NM92, Got09, Hoo96]. The happened-before relation shows how events potentially affect each other [Lam78], allowing developers to identify potential places that caused a bug and as such, offering a similar functionality as stack traces in sequential debuggers.

Although an event-based approach aids developers to detect the occurrence of a particular program (mis)behaviour, the examination of the corresponding computation states is typically still required to find the root cause of a bug. As such, some event-based debuggers combine post-mortem visualization of a program execution with online debugging features, e.g., Millipede, IDLI [BP06]. Others like Amoeba [Els89] combine replay with online debugging features such as breakpointing. In general, many of the event-driven debuggers focus on inter-process communication based on message passing and rely on a dedicated sequential debugger to deal with bugs internal to a process [CBM90].

### 9.1.3 Alternative Approaches

Many of the current distributed debuggers assume that applications run in a distributed object-oriented system and communicate by means of message passing. In this section, we look at a number of debugging techniques for alternative programming models such as actors and tuple spaces.

Honda et Yonezawa discuss in [HY88] several debugging and visualization tools for the actor-based language ABCL/1 based on the abstraction of *object groups*. Object groups provide an alternative approach to represent the behaviour of message passing programs by structuring event history in terms of the collective tasks performed by a group of objects. When one of the objects in an object group receives a message from an external object, it triggers a collective task for its group. The execution of such an entry message can in turn cause other message transmissions to other members of the group, denoting a task within the *object group task*. Object group tasks can be described best as a tree of objects whose root is the first object that received an entry message. The leaves of the tree then correspond to the objects which got a message but whose execution did not send any message.

IC2D [BBC$^+$01] is a graphical environment for monitoring and managing distributed ProActive applications (running on a grid). In order to monitor ProActive computations, it provides graphical visualisation including views to visualize the topology of active objects, and message sends and receipts for selected active objects. It also allows to interactively add a new or existing mobile active object to any running ProActive node as well as to move active objects to other nodes displayed by IC2D. A more recent implementation of the environment has been integrated within Eclipse IDE, allowing a step-by-step execution of an active object at the service level.

In [DS10], Dukielska et Sroka describe a distributed debugger for debugging tuple space-based applications written in JavaSpaces. The debugger combines online debugging features (such as breakpoints in tuple space operations) with replay functionality which allow developers to record and replay a sequence of tuple space operations.

## 9.2 Conclusion

Distributed debugging techniques and the debuggers developed to date have either been designed for parallel computing (e.g., p2d2, TotalView, Node Prism, Allinea DDT), for grid computing (e.g., Net-Dbx, and IC2D), or for general-purpose distributed computing in fixed, stationary networks (e.g., Amoeba, Causeway, and Millipede). None of these debuggers have been explicitly designed for applications running on mobile networks. They lack the necessary features to deal with the difficult task of debugging distributed asynchronous applications which run on a radically different network topology, in particular, to deal with the effects of partial failures. After all, debugging requires a thorough understanding of the application being debugged, as well as the software platform on which it is built. In the next chapter, we discuss the challenges of debugging ambient-oriented applications, and subsequently propose an ambient-oriented debugger whose goal is to provide a simple but useful debugging toolbox for ambient-oriented applications.

# Chapter 10

# Debugging in the Face of Partial Failures

Debugging software is an integral part of the development of any application. This task, which in sequential programs is already difficult, is further complicated in a distributed environment [CBM90]. When debugging a distributed program, developers must deal with the inherent non-determinism of concurrent processes. This complicates the debugging task since an error detected in one execution might not manifest itself in the debugging session. Furthermore, the mere presence of the debugger might exacerbate this non-determinism by affecting the way in which the program behaves. Computations performed by the debugger —recording state, checking for breakpoints, etc.— may affect the order in which processes are executed, making the reproduction of a rare erroneous condition even rarer. This condition akin to the Heisenberg uncertainty principle, is known as the *probe effect* [Gai85, MH89].

In this chapter, we focus on providing support for debugging applications running on a MANET. Since partial failures may percolate from the underlying distributed system layers up to the graphical user interface of an application, the need arises for managing partial failures up to the tool level. This observation led us to investigate debugging support for MANET applications together with the programming support for dealing with partial failures. We first detail the challenges of debugging MANET applications built around the principles of ambient-oriented programming, and present the main features of an *ambient-oriented debugger* to address them. We then describe the design and implementation of REME-D (read as remedy), a Reflective Epidemic MEssage-oriented Debugger. REME-D is an example of an ambient-oriented debugger for AmbientTalk that integrates techniques from traditional sequential debuggers (stepping and state inspection) and distributed debuggers (event-based debugging, message breakpoints) and proposes novel facilities to deal with ad hoc, fragile networks (epidemic debugging, and support for network disconnections). REME-D 's features have been implemented in AmbientTalk using the reflective architecture from Chapter 5.

## 10.1 Motivation

Before describing the features of an ambient-oriented debugger, we highlight the need for such a technique by discussing the challenges of debugging MANET applications.

To this end, we use an application scenario that we will also use as the running example throughout this chapter.

### 10.1.1 Running Example: the Mobile Shopping Application

Consider an adaptation of the scenario of the shopping application found in [SCM09] that runs on mobile devices. Users can launch the shopping application on their smartphone and add items in their shopping cart or check out when they are finished. When the user checks out the shopping cart, the application implements a protocol for handling purchase orders similarly to well-known shopping websites. Before the shop can acknowledge an order, it must verify three things: 1) whether the requested items are still in stock, 2) whether the customer has provided valid payment information and 3) whether a shipper is available to ship the order in time.

Figure 10.1 gives a graphical overview of the checkout protocol modelled via a distributed object-oriented system where communication between devices is asynchronous[1]. For simplicity, we use explicit callback objects to return the result of an asynchronous computation. When the user check outs the shopping cart in the shopping application UI, the `checkoutCart` message of the service object on the user's smartphone is sent which in turn sends the `go` message to the session object that was created for the user in a buyer process at the shop. In response to a `go` message, the buyer sends out three messages to the inventory, the credit bureau, and the shipper services called `partInStock`, `checkCredit` and `canDeliver` (as also shown in the figure).

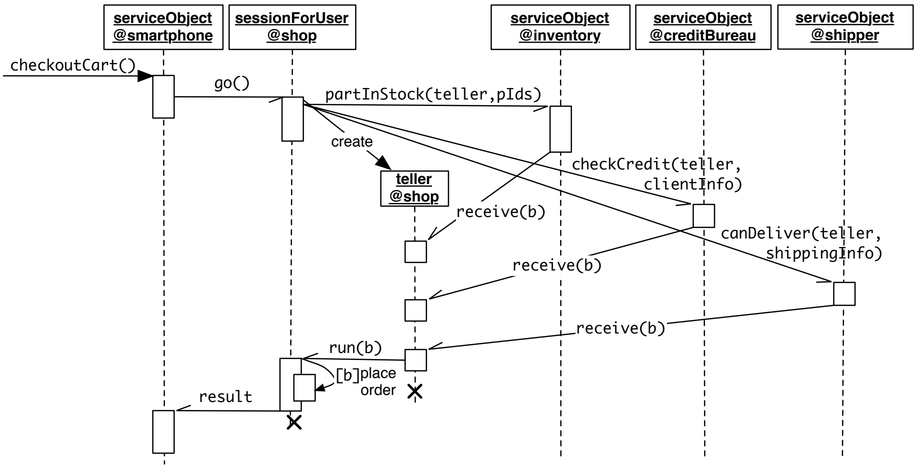


Figure 10.1: The shopping checkout protocol.

In order for the buyer object to collect the answer of the three services, a `teller` is created and passed as an argument in each of the above mentioned messages, serving as a callback object. A teller is an abstraction implementing an asynchronous adaptation of the logical `and` operator. The constructor of a teller object takes two parameters: a number indicating how many affirmative replies it should receive before it invokes `callback<-run(true)`, and the callback object to notify. The callback object thus

[1] To keep the figure concise and limited to the application functionality required for this chapter, we omitted the part of the protocol that actually places the order after all requirements are satisfied.

needs to implement the message `run(boolean)`. In this example, the teller is initialized to 3 replies, and the callback object to notify is the session object residing at the buyer. Hence, once the teller receives the three expected replies, it reports back to the shopping session `true` if all received replies were `true`; otherwise, `false`. The buyer then places the order only if all the requirements become satisfied.

## 10.1.2 Challenges of Debugging Ambient-Oriented Applications

In this work, we focus on providing debugging support for *ambient-oriented applications*: distributed applications running on MANETs that are built using a non-blocking concurrency model (as defined by the AmOP paradigm, cf. Section 2.2). Developing ambient-oriented applications is hard because of the effects engendered by partial failures and the fact that the network is open and has little or no infrastructure; developing debugging tools for such applications is even harder. As described in Chapter 9, a diverse spectrum of debugging tools exists for concurrent and distributed programs. However, they lack mechanisms to enable debugging in a mobile environment. To this end, two challenges need to be addressed which are discussed in the following sections.

### 10.1.2.1 Message-oriented debugging

In non-blocking concurrency models, non-determinism is limited to the order in which asynchronous messages are processed since a message is executed atomically. On the other hand, the distance between the cause of an error and its manifestation (i.e., error latency [CBM90]) can be larger. In sequential debugging, a call stack trace is often used to establish a *happened-before* relation [Lam78] between function calls. In a non-blocking concurrency model, at the beginning and end of processing each message, the call stack is always empty. This means that there is no trace of the path taken to reach the current execution point outside of a process; thus inter-process communication history is lost. It is precisely this inter-process communication that is essential to understand the behaviour of a distributed application. In our running example, the shopping checkout protocol behaviour consists of nine asynchronous messages, the most relevant being the three asynchronous messages encoding interactions with other processes to be verified before placing an order – `partInStock`, `checkCredit` and `canDeliver` messages. Consider a bug manifests itself when processing the `run` message in Figure 10.1. In order to find what caused the bug, one should start examining the `receive` message from the shipper because it denotes the request that `run` tried to satisfy. In the worst case, the entire “happened-before” relation chain must be considered, examining the `receive` messages from the credit bureau and inventory processes as well. In short, a debugger designed for a non-blocking concurrency model must be able to trace message passing between communicating parties, leading to the concept of *message-oriented debugging*.

### 10.1.2.2 Open debugging

To start a debugging session, a debugger is launched and then the target application is executed within the debugger environment. In traditional distributed debugging, the debugger may need to interact with the runtime system in order to manage the different processes which a distributed application consists of [WCS02]. Given the dynamic nature of MANETs, the number of processes that comprises an application is unknown when the debugging session is started. In our running example, we actually assume a nomadic network infrastructure in which the users are mobile while shop and other services are stationary computers that are interconnected via the nomadic network's infrastructure. However, an ambient-oriented application typically has to discover and collaborate with other partner applications when they meet in the environment as the device moves about. *As a result, a debugging session will consist of an undetermined, fluctuating number of processes according to the applications discovered in the network.* Because of this, a debugger must be able to dynamically add an application to an ongoing debugging session at runtime, i.e., the debugging session must be open. Furthermore, the debugger must also allow objects to leave the debugging session without affecting the rest of the participants as devices may leave the network at any time. Supporting open debugging sessions, along with a mechanism to detect and deal with the loss of participants in the debugging session should therefore be part of an ambient-oriented debugger.

Apart from supporting debugging in the context of a dynamically changing network topology, a debugger also needs to be able to engage in a deployed MANET application, i.e., an application which is already running on a mobile network when a debugging session starts. As Dao et al. remarked in [DAKV09], debugging a distributed application is challenging because the correctness of the system at a given point in time "depends on a combination of past and present system, node, and the network states". Reproducing the spectrum of possible states that a distributed application can be in and exposing the application to them before its deployment may be not feasible. As a result, many bugs may not manifest until the application is actually deployed. This is exacerbated in MANET applications since the correctness may also depend on the unpredictable mobility of the devices. In short, an AOD should also incorporate facilities in order to allow developers to debug "in situ" on a deployed mobile system (since discovered services can be added dynamically to the debugging session).

### 10.1.3 Summary

The above two characteristics have been distilled from the analysis of the implications of the hardware phenomena inherent to MANETs on the design of a distributed debugger. We will henceforth refer to distributed debuggers that adhere to them as *ambient-oriented debuggers* (AODs). AODs incorporate the effects engendered by partial failures and communication with anonymous discovered services in their design. In short, they should be able to deal with messages passed between communicating parties and provide control over the flow of asynchronous messages, as well as being able to be dynamically deployed on devices when necessary.

## 10.2 Overview of Ambient-Oriented Debuggers

Having defined the set of characteristics of ambient-oriented debuggers, we now describe the design and implementation of an ambient-oriented debugger which has been built around two central ideas: to adapt features from breakpoint-based debuggers to a non-blocking concurrency model (i.e., an event loop concurrency model), and to treat the debugging process as an ambient-oriented application which adapts its behaviour in order to respond to the openness of MANETs. Although the current prototype implementation of such an ambient-oriented debugger was done in AmbientTalk, its principles are independent of the implementation language and can be equally developed in other languages and software platforms for mobile networks built non-blocking concurrency models (e.g., in White [Qui07], or iScheme [BVB+12] (an Scheme dialect embodying the AmOP principles for iPhone application development), or an existing tuple space-based middelware).

A note on terminology before explaining the main core features and architecture of an ambient-oriented debugger in more detail: for simplicity, we will use the term ambient-oriented debugger to refer to an online ambient-oriented debugger. An online ambient-oriented debugger aims to reconcile the AmOP paradigm with a breakpoint-based debugging methodology. Although AODs may be incarnated by a post-mortem debugger, we have not investigated this in this work. The design and implementation of such an approach is outside the scope of this thesis.

### 10.2.1 Features of an Ambient-Oriented Debugger

We now introduce the key features of an AOD independently of its incarnation in AmbientTalk. It provides four major features: *state inspection, stepping, causal link browsing, and epidemic debugging*. The later is unique to ambient-oriented debuggers while the others are inspired by features of traditional debuggers. In particular, stepping and state inspection are typical features of sequential debuggers which are now adapted to a non-blocking concurrency model, while causal link browsing is a feature found in some post-mortem debuggers.We detail these features in the remainder of this section.

**State Inspection** An AOD is designed as a breakpoint-based debugger following a long-standing tradition of developing distributed debuggers based on sequential debuggers [DAKV09, Got09, NNE04, Hoo96]. A sequential debugger provides a user with visibility and control over the target application. AODs perform the same function for an ambient-oriented application. An ambient-oriented application consists of a number of processes (hereafter denoted as actors) running on various devices in a network. Each device typically runs a virtual machine hosting one or more actors that execute that part of the application, and communicate with each other by means of asynchronous message passing. As a result, an AOD is designed as a breakpoint-based debugger in which focus is placed on the exchange of asynchronous messages between actors.

An AOD supports the introspection of actors whenever they are suspended (in a pause state). An actor can be suspended *between* turns. Recall from Section 4.3, each method invocation corresponding to the processing of an asynchronous message spawns a turn which runs till completion (as a conventional sequential method invocation) before the next message is served. A turn may change the actor state and enqueue new messages to be delivered. Since turns are executed atomically, allowing actors only to be suspended between turns aligns well with a non-blocking concurrency model. In addition, the debugger's probe effect is minimized as turns in the actors of the debugged application remain atomic. This means that the debugger does not alter the way how a message is processed nor the order of outgoing messages sent as a result of its processing. When an actor is suspended, users can inspect the actor's state which consists of the objects hosted by the actor as well as the actor's mailbox.

**Stepping** To control the debugged application, users can place breakpoints to mark "interesting points" in the execution of the program at which the developer wishes to inspect the state. In a sequential debugger, breakpoints are placed on instructions, e.g., on the assignment of a variable or a method call. In message-oriented debugging, such interesting points take the form of messages exchanged between actors. As a result, in a non-blocking communication model, there are two places in which the debugger may check if a message hits a breakpoint: when the actor serves a message that needs to be sent to another actor (i.e., on the actor's outgoing message queue), and when the actor serves a message that needs to be received by one of its objects (i.e., on the actor's incoming message queue). We denote by *breakpointed message* a message which has hit a breakpoint and will pause the actor's execution when it reaches the head of an actor's message queue. We detail the different breakpoint semantics supported in our concrete incarnation of an AOD in Section 10.3.2.

The notion of applying breakpoints on messages maps well to a non-blocking concurrency model, since it allows defining meaningful stepping semantics at the message passing level. In AODs, stepping consists of executing the target application one *turn* at a time. As in a sequential breakpointed debugger, three kinds of step commands are offered: *step-over*, and *step-into* and *step-return* a turn. The concrete semantics of each step command is discussed later in Section 10.3.3.

**Causal Link Browsing** Causal link reconstruction allows the user to browse the history of messages sent and received in a turn. In sequential debuggers, the call stack gives the developer an idea of how the application has reached its current state. Unfortunately, in a non-blocking concurrency model, the call stack is empty at the end of each turn, thus providing no information to the debugger. Since all inter-actor communication is performed via asynchronous message passing, a traditional call stack is of no use in establishing the history of the distributed behaviour of the application. Rather, a partial order of messages sent and received would accurately reflect the distributed behaviour of the application. To this end, REME-D records the exchange of asynchronous messages during the execution of the debugged application, and then lets the user browse the obtained message trace.

A partial order of message sent and received provides an order of *activation* of computations similar to the call stack in sequential debugging. However, as Pothier et al. already remarked in [PTP07] "bugs often manifest themselves long after their root cause occurs". In sequential debugging, even if the execution is suspended before a fault, the root cause of a bug may already be lost, e.g., because the code that caused it is not accessible on the call stack anymore [PTP07]. In order to aid the user in the process of finding *when* the root cause of a bug occurred, AOD adopts an event-driven approach and also records the history of turns generated by the application. Such history represents the *process order* [Lam78], i.e., the order of message arrivals in an actor. In an AOD, a user can query the turn from where a message originated and the message that was being processed in that turn, thus establishing a *happened-before* relation between messages. The developer can then interactively unravel the *causal links* that led to the currently inspected message.

**Epidemic Debugging** Finally, one of the most distinctive features of AOD is its ability to respond to the dynamic nature of MANETs. Recall from Section 2.1 that any application deployed in a MANET is subject to two hardware phenomena, namely frequent disconnections and the lack of infrastructure. In order to deal with such hardware characteristics specific to the applications being debugged, AOD should itself be built as an ambient-oriented application which employs a non-blocking communication model to instrument the actors participating in a debugging session. This means that both the target application and the debugging infrastructure communicate by means of asynchronous message passing.

In addition, an AOD provides *epidemic debugging* which allows an actor which is discovered in the network to join an ongoing debugging session. Epidemic debugging allows the debugging support to be dynamically installed on newly discovered actors, a process akin to an infection in which the debugger spreads to devices joining the debugging session. As a result, applications can take part in a distributed debugging session without having to explicitly be configured as a participant beforehand. Devices can leave the debugging session at any point in time —either due to communication failures or in response to a user action— without disrupting the debugging of the remaining participants.

### 10.2.2 Architecture

Figure 10.2 gives an overview of the architecture of the ambient-oriented debugger. When debugging an ambient-oriented application, there may exist several devices running parts of the application. As such, debugging support will be distributed over two or more devices. In this case, Figure 10.2 shows three devices, two of which have joined the debugging session. The device which starts the debugging session of an application is called the *debugger device*. A device which joins the debugging session at a later point in time is called an *infected device*. To be more precise, an infected device is a device which contains at least one actor participating in the debugging session. Finally, there may be devices in the network which are conceptually running part of the target application, but which do not form part of the debugging session. This can happen because e.g., they are out of communication range of the debugger device, or because they do not run code relevant to the part of the application being debugged. A device can also opt out of being debugged, and as such, it will never be infected.

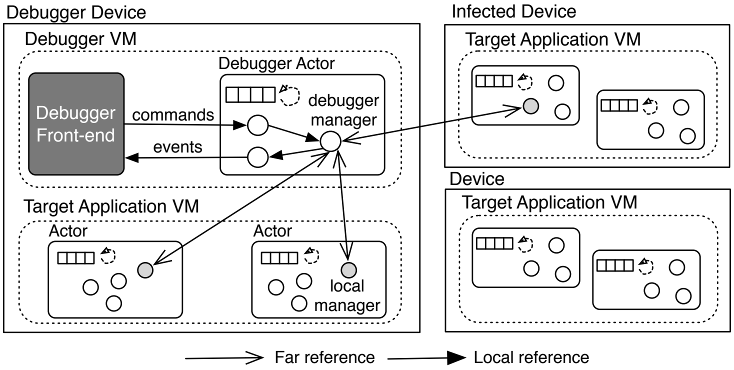


Figure 10.2: An ambient-oriented debugging session distributed over two devices.

As also shown in figure 10.2, the debugger device runs two virtual machines: the debugger virtual machine (VM) which hosts debugging infrastructure, and the virtual machine in which the target application runs. The debugger VM consists of two components: the coordinator debugger actor (or just *debugger actor*), and the debugger front end through which the user can interactively control the target application. Each actor participating in the debugging session contains a dedicated object (denoted in grey in the figure) called *local debugger manager* (or just local manager) that implements the main debugging features previously described. The debugger actor serves as a central manager between the debugger front end and all actors participating in a debugging session. Users can control a debugged application via the debugger front end, which issues debug commands (or just *commands*) in response to the user's actions (e.g., set a breakpoint, etc.). These commands are forwarded to the debugger manager, which in turn transmits it to the corresponding local manager(s). In response to those commands, a local manager will perform some action (e.g., pausing the actor execution) and inform the debugger actor of their state by sending debug events (or just *events*). The debugger actor forwards the events to the debugger front end so that it updates the debugging information presented to the user. As such, communication between the debugger manager and the local managers is bidirectional and happens via asynchronous message passing.

## 10.3 REME-D: an AOD in AmbientTalk

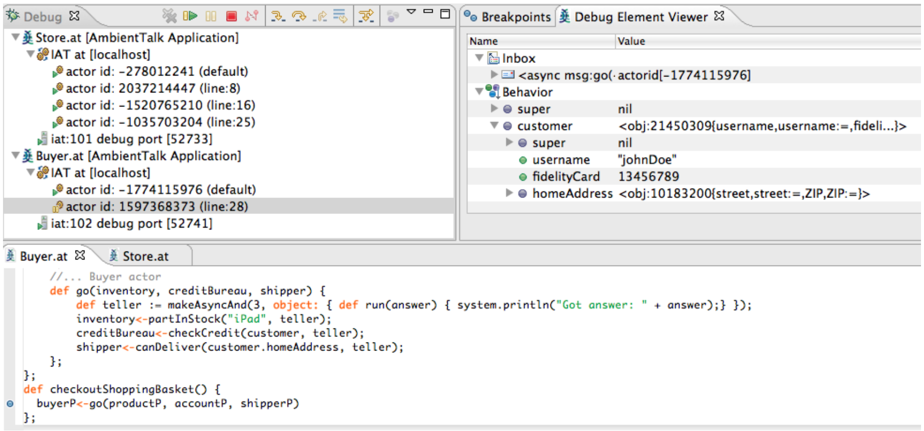
In this section, we describe the features of AODs within the context of a concrete incarnation of this concept in AmbientTalk called *REME-D*. REME-D is a Reflective Epidemic MEssage-oriented Debugger that has been implemented as the debugger module of the AmbientTalk IDE for Eclipse (IdeAT)[2]. REME-D implements AOD as an ambient-oriented application written on top of the reflective architecture introduced in Chapter 5. In this prototype, the debugger front end is implemented by a number of Java GUI components written on top of the Eclipse Debug Framework, and the debugger and the application VMs are implemented as AmbientTalk VMs. REME-D 's GUI displays the debugging information sent to it by participating distributed AmbientTalk VMs, and allows users to issue debug commands which are sent to the debugger actor via AmbientTalk's interoperability layer with Java (cf. Section 4.5).

Figure 10.3 shows a typical REME-D session in the context of the shopping application described in Section 10.1.1. The figure displays three views: the debug view in the top left pane (also called the *actor view*), the debug element viewer in the top right pane (also called the *actor state inspector*), and the editor at the bottom. As shown in the actor view, the application is composed of two different AmbientTalk VMs running the `Store.at` and the `Buyer.at` AmbientTalk files. The editor shows part of the implementation of the buyer actor which contains a `go` method. As previously explained, this method is called when the customer decides to purchase the goods in his shopping cart. Recall that in response to this action, the buyer actor consults three service providers (the inventory, the credit bureau and a shipper) in order to verify the order before placing it.

### 10.3.1 Viewing the Actor State

REME-D supports state inspection of actors whenever they are suspended (i.e., in a pause state). In particular, users can view the state of an actor reachable from the actor's *behaviour* object, and the messages that wait in the actor's message queue to be processed, i.e., the actor's incoming messages queue. In AmbientTalk, the actor's behaviour object denotes the first object created within an actor and represents the root (i.e., entry-point) for the rest of objects owned by the actor.

[2] The IdeAT plugin is available to be installed from the Eclipse update site at `http://tinyurl.com/ideat`, and its documentation is available at `http://tinyurl.com/ideatdocs`



```
Store.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -278012241 (default)
    actor id: 2037214447 (line:8)
    actor id: -1520765210 (line:16)
    actor id: -1035703204 (line:25)
  iat:101 debug port [52733]
Buyer.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -1774115976 (default)
    actor id: 1597368373 (line:28)
  iat:102 debug port [52741]
```

| Name | Value |
|---|---|
| Inbox | |
| <async msg:go(· | actorid[-1774115976] |
| Behavior | |
| super | nil |
| customer | <obj:21450309{username,username:=,fideli...}> |
| super | nil |
| username | "johnDoe" |
| fidelityCard | 13456789 |
| homeAddress | <obj:10183200{street,street:=,ZIP,ZIP:=}> |

```
    //... Buyer actor
    def go(inventory, creditBureau, shipper) {
        def teller := makeAsyncAnd(3, object: { def run(answer) { system.println("Got answer: " + answer);} });
        inventory<-partInStock("iPad", teller);
        creditBureau<-checkCredit(customer, teller);
        shipper<-canDeliver(customer.homeAddress, teller);
    };
};
def checkoutShoppingBasket() {
  buyerP<-go(productP, accountP, shipperP)
};
```

Figure 10.3: Eclipse plugin showing a REME-D debug session.

While an actor's execution is paused, the state of its objects remains static —no other execution thread has access to them— and no messages are added to the actor's outgoing message queue —the paused thread is the only execution thread in an actor. However, non-paused actors can still send messages to it. New messages that arrive while the actor is paused are enqueued, and the local manager notifies REME-D's UI that the state of the actor's message queue changed. Note that while an actor is paused, its incoming message queue can only grow.

Actors can be explicitly suspended by the developer via a pause command, or implicitly suspended by the local manager as a result of a breakpoint or a step command. When an actor is suspended, the corresponding local debug manager delays the processing of the message at the head of the queue, until it receives the command to resume execution. The local manager communicates the state of the actor to the debugger actor which in turn updates REME-D's UI. Figure 10.3 shows how this information is presented to the developer in REME-D in the state inspector. The developer is able to inspect the state of the objects encapsulated in the actor, as well as the messages that await processing in the actor's incoming message queue. In this example, the actor contains a `customer` and a `shoppingCart` object, and a `go` message emitted by an actor with the id `-1774115976`. By default, REME-D's UI shows all objects that exist in the behaviour object's scope in a list, and then lets the user navigate the object graph by unfolding the object to be inspected. For example, Figure 10.3 also shows the state of the `customer` object which consists of three fields, a username, `fidelityCard` number and a `homeAdress` object (which is currently folded).

## 10.3.2 Breakpoints Catalog

As explained in Section 10.2.1, an AOD must allow users to set breakpoints in asynchronous messages, as opposed to instruction breakpoints in traditional sequential debuggers. REME-D provides a catalog of breakpoints which combines breakpoints on messages with some characteristics of traditional breakpoints in sequential debugging. More concretely, the breakpoints supported in REME-D can be classified according to three basic properties, leading to the elementary breakpoint types shown in the Figure 10.4.

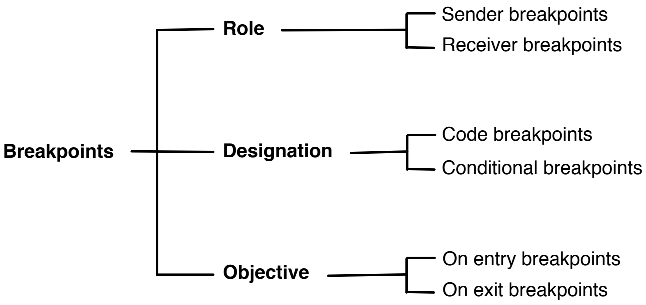


Figure 10.4: REME-D elementary breakpoint types.

**The role of a breakpoint** determines the place at which the breakpoint is set. As already mentioned in Section 10.2.1, we distinguish between breakpoints placed in the actor's outgoing message queue (called *sender breakpoints*), and actor's incoming message queue (called *receiver breakpoints*). In the remainder of this chapter, we refer to the actor's incoming message queue as just the actor's message queue, and the message queue's full names will be used to clarify the explanation when necessary.

**The designation of a breakpoint** denotes the way that a user defines a breakpoint. We distinguish between breakpoints defined in terms of a line of code (called *code breakpoints*), or in terms of a predicate condition about the state of a message (called *conditional breakpoints*).

**The objective of a breakpoint** denotes when the execution should be suspended. We distinguish between breakpoints which suspend the actor's execution when a message reaches the head of the message queue, (a) before the message is executed (called *on entry breakpoints*) and (b) after the message is executed (called *on exit breakpoints*). REME-D provides users with 5 different combinations of these elementary breakpoint types which are described below:

**Message breakpoints** A message breakpoint defines a breakpoint on a line of code of an asynchronous message send. Recall from Section 4.3.1 that in AmbientTalk an asynchronous message is expressed as `o<-m()` while a synchronous one is expressed as `o.m()`. In Figure 10.3 this is indicated in the editor by a blue dot next to the `go` asynchronous message. The figure shows the buyer actor paused as a result of this breakpoint, and one can see the `go` at the head of its message queue in the state inspector view on the right top corner. The actor's execution pauses when the breakpointed message reaches the head of the message queue, *before* the receiver invokes the method corresponding to the asynchronous message.

**Message resolution breakpoints** A message resolution breakpoint defines a breakpoint on a line of code of an future-type message send, typically expressed as `o<-m()@FutureMessage` in AmbientTalk. The actor execution pauses when the message carrying the return value of the computation reaches the head of the message queue of the sender actor. This means that execution is paused after the future-type message is processed but, *before* the sending actor processes the message with the return value of the computation.

Recall from Section 4.3.2 that an AmbientTalk asynchronous message does not have a return value, but immediately returns **`nil`**. Thus, setting a message resolution breakpoint on an regular asynchronous message send which is not futurized, does not have any effect since no actor will be stopped ever. For example, placing a message resolution on the `go` asynchronous message in our running example will never pause the actor running the application on the end-user device.

**Method breakpoints** A method breakpoint defines a breakpoint on a line of code of a method definition. The actor's execution pauses when *any* asynchronous message invoking the method defined on the given line of code reaches the head of the actor's message queue.

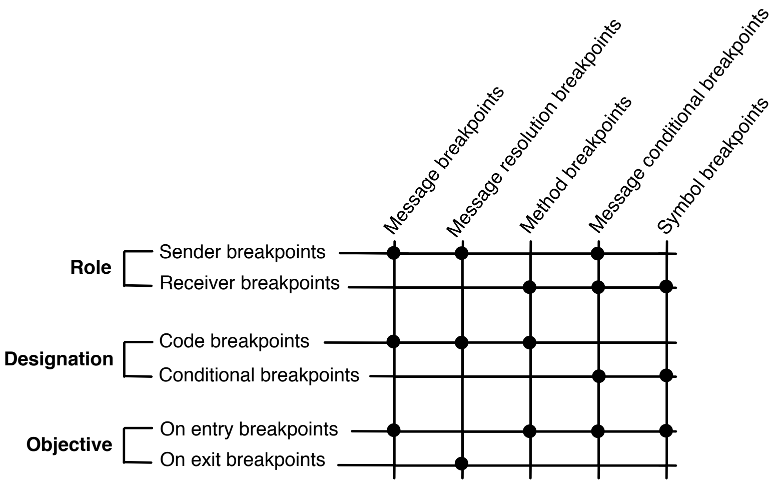
Note that in our running example, setting a breakpoint on the method definition of the `go` method in the buyer is equivalent to setting a message breakpoint on the `go` asynchronous message since the `go` method is usually called only once on a shopping session that the application on the end-user smartphone has opened on the buyer process.

**Symbol breakpoints** A symbol breakpoint defines a breakpoint on a method name corresponding to a given symbol. The actor's execution pauses before the receiver object invokes a method whose name is the given symbol. This is signaled in the editor by selecting a symbol and toggling a breakpoint. A symbol breakpoint has a similar effect as a method breakpoint but it may pause the actor's execution before invoking the method on *many* receivers since there may exist different objects defining a method with the same name. For example, recall again the the mobile music player application described in Section 6.1.1. Many copies of the song prototype object (cf. Listing 5.5) can coexist in the system, all implementing the same API. One can imagine that a user may want to place a symbol breakpoint for a method name ( e.g., `play`) to investigate which songs may have been wrongly received and cannot be played.

**Message conditional breakpoints** A message conditional breakpoint defines a breakpoint on a conditional expression about a message. It works similarly to a *watchpoint* for a message: it allows users to stop execution whenever the result of an expression is true, without having to predict a particular message send or reception where this may happen.

Recall from Section 4.6.1, that in AmbientTalk, an actor's message queue actually consists of a receiver-message pair (called a letter). Hence, the conditional expression is a predicate on the state of the message-receiver pair. It can contain arbitrary AmbientTalk code including more than one statement, but the return value of the expression must be a boolean. A user can select in which message queue the conditional breakpoint should be placed by means of the "breakpoint properties..." dialog. The actor's execution pauses when the result of the conditional expression is true for a message that reaches the head of the given message queue(s), before the message is processed.

Figure 10.5 provides an overview of these classes of combined breakpoints according to the taxonomy shown in Figure 10.4. In addition to such a breakpoint catalog, the implementation of REME-D itself employs two other combinations that we describe later in Section 10.4. Note that more combinations may be identified and added in the future using the debugger breakpoint abstraction (also described in Section 10.4).



| | | Message breakpoints | Message resolution breakpoints | Method breakpoints | Message conditional breakpoints | Symbol breakpoints |
|---|---|---|---|---|---|---|
| Role | Sender breakpoints | ● | ● | | ● | |
| | Receiver breakpoints | | | ● | ● | ● |
| Designation | Code breakpoints | ● | ● | ● | | |
| | Conditional breakpoints | | | | ● | ● |
| Objective | On entry breakpoints | ● | | ● | ● | ● |
| | On exit breakpoints | | ● | | | |

Figure 10.5: REME-D 's breakpoint catalog provided to users.

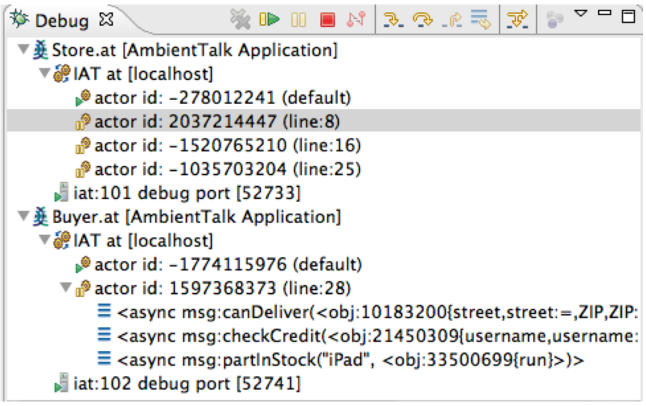
When a user sets a breakpoint, the debugger actor notifies all local managers of the new breakpoint so that they can update their internal data structures and initialize the necessary machinery to pause the actor's execution accordingly. Note that peer-to-peer is a recurring pattern in ambient-oriented applications in which a single object plays the roles of both client and server. Since all peers share the same source code, the debugger should provide a mechanism for specifying on which device the breakpoint should be active. REME-D enables control over the activation of breakpoints by allowing users to select at runtime on which devices a given breakpoint should be active, similarly to the support provided in p2d2 [Hoo96]. This is set by means of the "breakpoint properties..." dialog that appears when right clicking on the dot indicating a breakpoint. REME-D UI will then contact the debugger actor which, in its turn, notifies the corresponding set of local managers of the (de)activation of the breakpoint.

### 10.3.3 Stepping

REME-D allows users to perform a step-by-step execution of a running application. As in a sequential breakpointed debugger, three kinds of step commands are offered: step-over, and step-into and step-return a turn. In addition, we provide a variation of step-over called step-until. In the remainder of this section, we describe those commands.

**Step-Over Command** Stepping over a turn allows the user to observe how the state of the actor changes as it processes incoming messages. A step-over command instructs the local manager to process a single message —the one at the head of the queue— and returns to the paused state. In addition, the local manager keeps track of all outgoing messages that should be sent during that turn, allowing the user to inspect them.

**Step-Into Command** Stepping into a turn allows the user to navigate the *consequences* of processing a given message, i.e., the messages sent to other actors in that turn. This is important when understanding the behaviour of an actor since actors co-operate in order to carry out tasks [HY88]. When the user instructs REME-D to step



```
▼ Store.at [AmbientTalk Application]
  ▼ IAT at [localhost]
      actor id: -278012241 (default)
      actor id: 2037214447 (line:8)
      actor id: -1520765210 (line:16)
      actor id: -1035703204 (line:25)
    iat:101 debug port [52733]
▼ Buyer.at [AmbientTalk Application]
  ▼ IAT at [localhost]
      actor id: -1774115976 (default)
    ▼ actor id: 1597368373 (line:28)
        <async msg:canDeliver(<obj:10183200{street,street:=,ZIP,ZIP:
        <async msg:checkCredit(<obj:21450309{username,username:
        <async msg:partInStock("iPad", <obj:33500699{run}>)>
    iat:102 debug port [52741]
```

Figure 10.6: Debug view after a step-into command.

into the current turn, the local manager will perform a step-over and mark all outgoing asynchronous messages as breakpointed messages. As a result, at the end of step-into, the current actor (i.e., the one on which the command was invoked) and all the actors receiving messages sent on that turn are paused. The user can then assert the effect of the turn on the actor's state, and decide which of the now paused recipient actors he wants to continue debugging. This semantics is reminiscent of the semantics provided in [Wis97] by the combination of a message breakpoint with a traditional breakpoint on the send statements.

Figure 10.6 shows the debug view after having stepped into the turn that processed the `go` message shown in Figure 10.3. The actor that performed the turn is expanded displaying the list of messages emitted during the turn processing `go` message (`canDeliver`, `checkCredit` and `partInStock`). The three actors that received them are now paused as indicated by the yellow pause signs.

**Step-Return Command** Step-returning a turn allows the user to return from a message which has been stepped into. A step-return command is really useful when debugging future-type asynchronous message sends. When the user instructs REME-D to step return from a future-type message, the local manager will perform a step-over of the message, and mark the message sent with the return value of the computation as a breakpointed message. As a result, at the end of step-return of a future-type message, the actor that *sent* the message is paused when the future associated with the message becomes resolved. The user can then inspect the return value of the message sent, and can decide to continue debugging from the start of any *callback* functions registered with the future by means of the **`when:becomes:`**$*$ functions (cf. Section 4.3.2).

**Step-Until Command** In addition to stepping over a turn, REME-D also allows to step over several turns until a certain message reaches the head of the actor' message queue. A step until command takes a conditional expression about a message (reminiscent of message conditional breakpoints), and instruments the local manager to step over the execution of a number of messages, pausing the execution again when a message that satisfies the given condition is found. A step-until command is specially handy when debugging distributed interactions in which the same message is sent several times to an object with different state (e.g., taking different argument values). For example, in our running example, the service object in the customer's smartphone sends a `addItemToCart` message each time the customer adds items to the cart of the shopping session with the buyer process. One can imagine that if a bug manifests itself in the shopping protocol, a user may want to instrument the debugger to make a step over all `addItemToCart` messages until the one suspected to be the root cause of a bug reaches the head of the message queue, e.g., the last `addItemToCart` message, or one adding a particular sort of items.

### 10.3.4 Browsing Causal Links

Stepping allows users to interactively move forward in the computation in order to determine whether it behaving as intended. To assert the cause of a bug, it is also important to know what has happened earlier in the computation. As we explained, in an ambient-oriented debugger, the history of messages sent and receptions provides users with a partial causality similar to the call stack in sequential debugging. Such information about the communication traces of actors has also proven to be helpful to implement a variety of synchronisation mechanisms, leading Dedecker to state it as a dedicated requirement in his original formulation of the AmOP paradigm [DVM$^+$06].

In REME-D, the message trace is contained within the message itself. Each message contains debugging information about the trace of messages from which the message originated. A trace of messages consists of a list of $< identifier, selector >$ tuples which contains the identifier of the turn in which the message send got created, and the selector of the message being processed in that turn. When a message gets sent in a turn, the local manager attaches the list of tuples for the message being processed to the message, and then adds a new tuple with the debugging information for that turn. For example, in our running example, the `canDeliver`, `checkCredit` and `partInStock` messages were sent during the turn in which the `go` message was processed. The local manager in the buyer actor will extend those three messages with the message history that the `go` message already had, and a new tuple including the identifier of the turn processing `go`, and the `go` selector. The user can then ask in REME-D's UI to inspect the message history for a given message. For example, the message history of `canDeliver` message shows debugging information about the `go`, and `checkoutCart` messages.

As an exemple of an ambient-oriented debugger, REME-D allows users to query the turn from which a message originated. Local managers maintain the history of turns executed by the actor, noting the incoming and all outgoing messages in the turn. When an actor processes a message from the message queue, the local manager stores a $< id, cause, effects >$ tuple which contains the turn identifier, the selector of the message being processed (representing the *cause* of the turn), and the list of all outgoing messages sent during the turn (representing the *effects* of the turn). This information is then sent to the debugger actor when a user would like to browse the turn for a particular message being inspected.

### 10.3.5 Open Debugging

In this section, we detail how REME-D deals with the hardware characteristics of MANETs, in particular, how it supports debugging in the face of partial failures of the target application. REME-D has been designed as an ambient-oriented application in which communication is non-blocking. This means that REME-D sends debug commands and receives events from participating actors via asynchronous message passing. When a local manager detects a communication failure with the debugger manager, it removes all breakpoints and resumes the actor if necessary.

Since network disconnections are taken into account in the design of ambient-oriented applications, it is important to debug their behaviour in the face of these events. REME-D provides the user with the possibility of simulating the disconnection of a device. This is achieved by having the local manager on the “disconnected” device cut communication with other devices, while maintaining it with the debugger manager (so that the user can still instrument the actor from REME-D’S UI).

#### 10.3.5.1 Epidemic Debugging

As previously mentioned, debugging applications written for MANETs require that the debugging process itself be open. As such, REME-D ’s debugging sessions are not constrained to a fixed configuration. The user does not need to define a-priori which devices will participate in the session. Instead, REME-D operates in an epidemic fashion, spontaneously adding devices to the current debugging session whenever they interact with actors participating in the session. More concretely, a device is added to the debugging session whenever it receives a breakpointed message from an actor. Upon receiving a breakpointed message, the AmbientTalk VM deploys a local manager on the receiver actor and the VM is said to be *infected*. The local manager then announces its presence to the debugger manager, which adds the actor information to the debugging session and sends back debugging information (e.g., the active breakpoints).

Recall from Figure 10.6, that the debug view on the right pane shows two different devices running `Store.at` and the `Buyer.at` AmbientTalk files, respectively. REME-D’S UI also displays newly infected devices as part of the debugging session in the debug view.

The infection of an actor only happens if the source file was loaded in an AmbientTalk VM started with `-Xdebug` option (akin to the “-g” option found in Java and C compilers), or if an actor dynamically activates debugging facilities (explained later in Section 10.4.4). Allowing devices to explicitly opt-out of a debugging session prevents the most obvious security issues. In this case, the `Store.at` file was executed within Eclipse with a custom run configuration. However, the file could have been executed outside of Eclipse, e.g., by means of the interactive AmbientTalk shell on a computer, or the AmbientTalk library for Android devices.

An actor may leave a debugging session in two cases: when the device hosting the AmbientTalk VM disconnects from the network, or because the user stops the debugging process on the AmbientTalk VM. If a device was suspended when it disconnected, the local manager resumes the actor. On the other side, the debugger manager removes the disconnected actor from the debug view. The debug view is thus updated whenever a device appears or disappears on the network.

## 10.4 Implementation

As previously mentioned, REME-D has been implemented in the AmbientTalk language itself, on top of the reflective infrastructure described in Chapter 5. The debugger manager has been implemented as a regular AmbientTalk object while the local manager is an actor mirror that alters the default semantics for message sending and

```
1  def debuggerEventListener := object:{
2    def startActorEvent(actorId, vmId, fileName, line){
3      eclipsePlugin.handleEvent(StartActorEvent.new(actorId, vmId, fileName, line);
4    };
5    def pauseActorEvent(actorId, actorState){
6     eclipsePlugin.handleEvent(PauseActorEvent.new(actorId,actorState));
7    };
8    /** rest of events **/
9  };
```

interface with debugger manager

```
10 def commandListener := object: {
11   def executeStartCommand(startCommand){
12     startCommand.getBreakpoints().each: {|bP|
13       bP.executeCommand(self);
14     };
15     debuggerManager.loadMainCode(startCommand.getActorId());
16   };
17   def executeSetBreakpointCommand(fileName, lineNumber, bpActiveOnList){
18     if: ( nil == bpActiveOnList) then: {
19       debuggerSessionBhv.setBreakpoint(fileName, lineNumber);
20     } else: {
21       debuggerSessionBhv.breakpointActiveOn(fileName, lineNumber, bpActiveOnList );
22     };
23   };
24   /** rest of commands **/
25 };
```

interface with Eclipse plugin

```
26 def debuggerManager := debuggerModule.makeDebuggerManager();
27 def eclipsePlugin := jlobby.edu.vub.at.debug.core.DebugCorePlugin.getDefault();
28 debuggerManager.setupDebugSession(debuggerEventListener);
29 eclipsePlugin.registerController(commandListener);
```

setup

Figure 10.7: Behaviour of the debugger actor.

reception. In the rest of this section we describe the implementation of the debugger manager, the local manager and other relevant abstractions such as breakpoints.

### 10.4.1 Creating and Managing A Debugging Session

As previously explained, the debug manager coordinates a debugging session. Similarly to the architecture shown in Figure 10.2, the debug manager is an object within an AmbientTalk actor called the *debugger actor*. Figure 10.7 provides an implementation overview of the debugger actor organized in three different parts[3]. First, the debugger actor defines two other objects which mediate communication between the debugger front end and local managers participating in the debugging session. The *command listener* object serves as the interface with the Eclipse UI and is called directly from the front end in response to the user's actions (e.g., set a breakpoint). The *event listener* object serves as interface with the debugger manager and is called to communicate updates to the debugger's front end (e.g., mark on the GUI that an actor was paused as a result of a breakpointed message). Finally, the debugger actor creates the debugger manager and sets up the bidirectional communication with the Eclipse UI (lines 26-29).

In response to a user's action, the Eclipse UI contacts the command listener using AmbientTalk's interoperability layer with Java. The command listener sends a message to the debugger manager for a given command which in turn communicates the command to the corresponding local manager(s). In response to debug commands, local

[3] For conciseness, we only show the implementation of a couple of command and events; the rest are implemented based on the same strategy.

managers may communicate changes about the actor/debugging state to the debugger manager which in turn reports them to the event listener. This implementation strategy decouples the core functionality of the debugger (provided by the debugger and local managers) from the front-end interactions (encapsulated in the command and event listeners). As a result, it eases the development of new debugger front ends since plugging new GUI front ends only requires modifying the event listener. In addition, it allows creating unit tests for the debugger core in the same way as they are implemented for regular AmbientTalk applications.

For the sake of reproducibility, Appendix C describes the interactions between debugger manager and command and event listeners for REME-D's startup protocol.

#### 10.4.1.1 The Debug Manager Object

Figure 10.8 gives an overview of the debugger manager. A debugger manager object is created by means of the `makeDebuggerManager` function which consists of three parts (delimited with an inner box in the figure). First, the function defines the necessary data structures to manage the debugging session. A hashmap `actorList` maintains an up-to-date list of connected actors in the debugging session. It is updated whenever an actor loses connectivity by registering a **`whenever:disconnected:`** handler (cf. Section 4.4.2). The hashmaps `breakpointsInAllVM` and `breakpointsInVM` store the breakpoints that should be active in all AmbientTalk VMs and only in a subset of AmbientTalk VMs, respectively. The debugger manager keeps those hashmaps up-to-date according to the user's actions on the UI, and informs the corresponding local managers when that is necessary. Hence, a local manager only receives information regarding to the breakpoints relevant to it (it does not known in which other VMs or actors the breakpoint is active). The hashmap `vmIdToActorIdMap` stores the actor identifier per AmbientTalk VM and it is used by `breakpointActiveOn` for computing the breakpoints active on a given list of VM identifiers. Finally, the `debugEventListener` stores a reference to the event listener to communicate changes reported by local managers to the debugger front end.

As shown in Figure 10.8, the `makeDebuggerManager` function also creates two interface objects: a `localInterface` object which defines all methods that can be invoked by the debugger front end via the command listener object, and a `remoteInterface` object which defines all the methods that can be invoked remotely by local managers to update the debugger front end. As explained before, when the command listener invokes one of the `localInterface` methods for a command, the debugger manager sends an asynchronous message to the corresponding local manager(s). Those messages are annotated with a `@Debug` annotation so that a local manager can distinguish between application-level messages and debug-level messages. Table 10.1 provides an overview of all annotations used in the REME-D prototype.

### 10.4.2 Instrumenting Actors In The Debugging Session

In order for REME-D to control an actor in the debugging session, a local manager is installed in each actor created during the debugging session. A local manager is an implicit actor mirror that alters the default language semantics for the message invocation protocol (described in Section 4.6.3) to implement the features of AODs.

```
def makeDebuggerManager() {
  def actorList := HashMap.new();
  def breakpointsInAllVM := Vector.new();
  def breakpointsInVM := HashMap.new();
  def vmIdToActorIdMap := HashMap.new();
  def debugEventListener;

  def localInterface := object: {
    def codeBreakpointActiveOn(filename, lineNumber, listVMIds){...};
    def clearCodeBreakpoint(filename, lineNumber){...};
    def setCodeBreakpoint(filename, lineNumber, messageResolved := false){...};
    def stepReturn(actorId){...};
    def stepOver(actorId){...};
    def stepInto(actorId){...};
    def resumeActor(actorId){...};
    def pauseActor(actorId){...};
    def loadMainCode(actorId){...};
    def setupDebugSession(debugEL){
      debugEventListener := debugEL;
      export: remoteInterface as: debuggerUtilModule.DebuggerManager;
      network.online();
    };
    def getLocalManagerById(localManagerId){...};
    def listLocalManagers(){...};
  };

  def remoteInterface := object: {
    def actorStarted(actorId, sourceLocation, frLocalManager){...};
    def actorPaused(actorId,actorState){...};
    def actorResumed(actorId){...};
    def updateInbox(actorId, msg, addition := true){...};
    def updateMessageSent(actorId, msg){...};
  };

  localInterface; //return value of this function
};
```

data structures

interface with front end

interface with local managers

Figure 10.8: The debugger manager object.

Figure 10.9 shows the definition of a local manager's implicit actor mirror. We explain later in this section how and when a local manager is installed on an actor. The `makeLocalManager` function creates a local manager by extending a given actor mirror with REME-D's debugging functionality. The function also takes as parameter a boolean denoting whether the local manager is joining a running debugging session and the debugger manager of that session (if known). We can distinguish four different parts in the behaviour of a local manager (delimited by an inner box in the figure).

| | |
|---|---|
| `@Debug` | Annotation used to mark messages as debugging commands from the debugger actor. |
| `@Pause` | Annotation used to mark messages that require pausing the receiver actor. It is used both by breakpointed messages and messages sent during step-into command. |
| `@PauseResolve` | Annotation used to mark messages that require pausing the sender actor. It is used both by messages breakpointed as a result of a message resolution breakpoint and messages sent during step-return command. |

Table 10.1: Annotations on asynchronous messages.

```
def makeLocalManager(actor, debuggingSession, debuggerManagerFarRef){
  extend: actor with: {
    def debuggingState := INITIAL;
    def pausedState := INITIAL;
    def debuggerManager := nil;
    def actorId := debuggerUtilModule.generateRandomId();
    def inbox := [];
    def senderBreakpoints := HashMap.new();
    def receiverBreakpoints := HashMap.new();
    def turnHistory := Vector.new();
    def turnId := 0;

    def send(rcv, msg) {/*explained later*/};
    def schedule(rcv, msg){/*explained later*/};
    def serve(){/*explained later*/};

    def remoteInterface := object: {
      def evaluateCode(){...};
      def startInDebugMode(tableCodeBreakpoints){...};
      def pause(){...};
      def resume(){...};
      def stepCommand(stepTypeTag) {...};
      def addBreakpoint(breakpoint){...};
      def removeBreakpoint(breakpoint){...};
    };

    def whenDiscoveryActorDiscovered(da) { /*explained later*/};
    if: (debuggerManagerFarRef == nil) then: {
      when: debuggerUtilModule.DebuggerManager discovered: { |da|
        whenDiscoveryActorDiscovered(da);
      };
    } else: {
      whenDiscoveryActorDiscovered(debuggerManagerFarRef);
    };
};
```

data structures

meta methods

interface with debugger manager

service discovery

Figure 10.9: Implicit actor mirror on the local manager.

**Data Structures** A local manager keeps track of a number of data structures. The `debuggingState` variable indicates whether the actor execution is in an initial, a running or a paused state. Once a local manager has been created and installed on an actor, the local manager is said to be in the initial state, but it has not started until it receives the `startInDebugMode` message from the debugger actor (as described in Figure C.1). Once started, the actor being controlled by a local manager may be running or paused. The `pauseState` variable indicates the reason to pause an actor's execution. An actor may be paused as a result of a pause command, a breakpointed messages, or any of the supported step commands. The `inbox` stores the base-level messages sent to an actor while it is paused. The `senderBreakpoints` and `receiverBreakpoints` hashmaps store the breakpoints that need to be checked when sending or receiving a message, respectively. Finally, the `turnHistory` vector keeps a list of objects representing the turn information, and the `turnId` variable stores the current turn identifier.

**Meta Methods** A local manager overrides a number of meta actor methods to alter the default language semantics for the message invocation protocol. We describe three changes to the semantics. First, the local manager's implicit mirror overrides the `schedule` meta method to check if a message hits a receiver breakpoint and pauses the actor's execution if necessary. Second, any asynchronous message sent during a turn is extended to include information about the turn and sender in order to build the event history for browsing causal links. This is done by overriding the `send` meta method.

`send` also checks if a message hits a sender breakpoint, in which case the message is annotated with a `@Pause` annotation. As described in Table 10.1, this annotation pauses the receiver actor. Finally, the `serve` meta method is overridden to implement the resume and step commands.

**Interface with the debugger manager** The `remoteInterface` object defines all the methods that can be invoked remotely by the debugger manager to instrument the actor's execution.

**Service Discovery** Finally, a local manager extends the default meta-actor protocol with the `whenDiscoveryActorDiscovered` method which takes care of sending the `actorStarted` event from the startup protocol explained in Figure C.1, and places the necessary failure handling code to deal with disconnections of the debugger manager. As shown in Figure 10.9 (lines 25-31), the `whenDiscoveryActorDiscovered` method is called when a debugger manager is discovered on the network, or it is immediately called if the debugger manager is already known (e.g., when infecting a new actor, the local manager is initialized with a given debugger manager as we discuss later in Section 10.4.4.

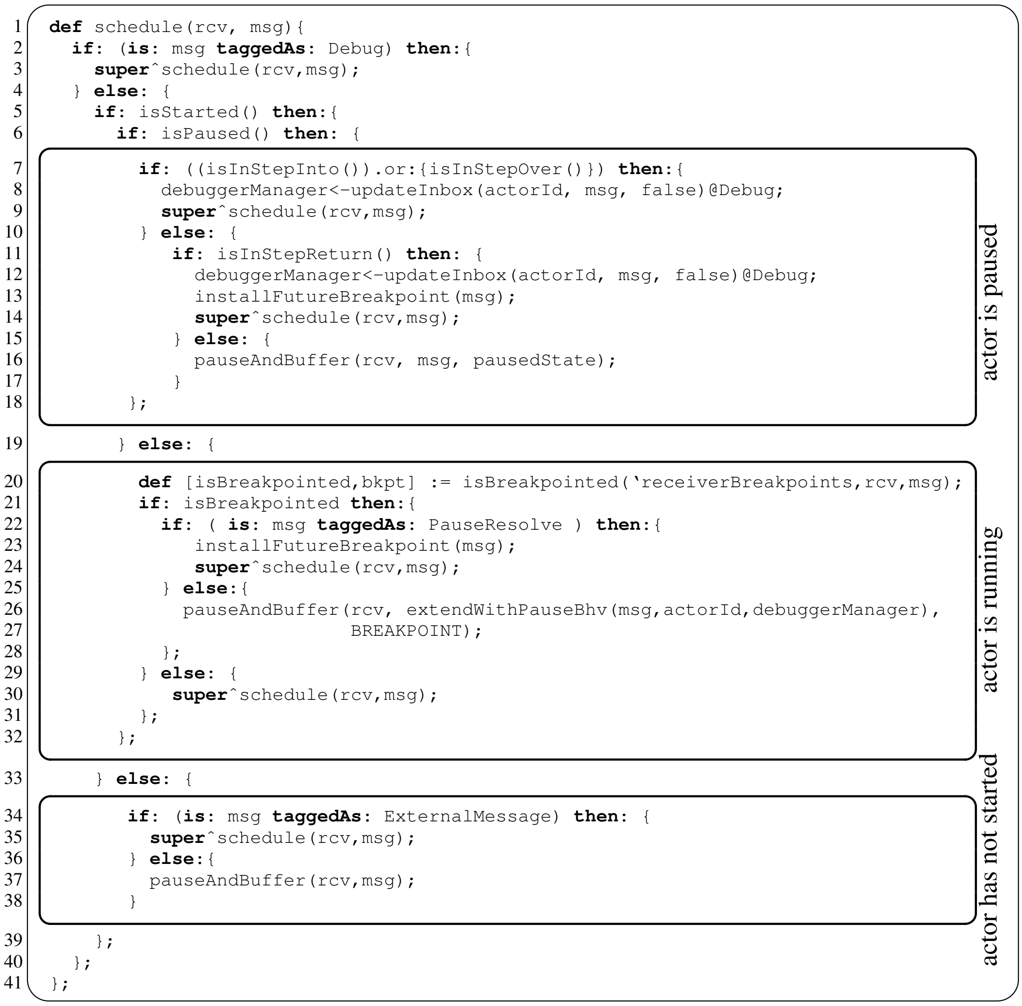
In the remainder of this section, we provide more insight about the three meta methods that the local manager's actor mirror overrides as they implement the core debug functionalities.

#### 10.4.2.1 Instrumenting Message Reception

Recall from Section 4.6.3 that the `schedule` method is called right before a message is added to the message queue of an actor. Figure 10.10 shows the behaviour of the `schedule` method in the local manager's implicit mirror. `schedule` first checks the message's annotations. If the message has a `@Debug` annotation, the default semantics of `schedule` are applied (lines 2-4) as it represents a debug-level message sent by the debugger actor. For all other messages, the local manager checks the debugging state to determine how to handle the message. Each debugging state is delimited by an inner box in the figure. When an actor receives a message when it has not yet started (lines 34-38), it will buffer all application-level messages that it receives from other actors until it receives the `startInDebugMode` message from the debugger actor. When an actor receives a message while running (lines 20-32), the local manager checks whether the message hits a breakpoint in the `receiverBreakpoints` hashmap, and pauses the execution if a match is found. Finally, if the actor is paused when `schedule` is called, e.g., due to a pause command (lines 7-18), the incoming message is buffered and the debugger actor is notified of the arrival of a new message. The debugger actor in turn updates the UI representation of the message queue in the inspector. These semantics are not applied for debug-level messages.

#### 10.4.2.2 Instrumenting Message Processing

Recall from Section 4.6.3 that the `serve` method is called when a message is dequeued, before being processed. It is overridden to create an object with turn information and adds it to the turn history for the message to be processed. The `serve` method also updates the `pauseState` variable after having served a message, i.e., when a turn is completed. If the message was executed as a result of a step-over or step-into command, the pause state is set to initial so that the actor's execution is again paused in the



```
def schedule(rcv, msg){
  if: (is: msg taggedAs: Debug) then:{
    super^schedule(rcv,msg);
  } else: {
    if: isStarted() then:{
      if: isPaused() then: {
        if: ((isInStepInto()).or:{isInStepOver()}) then:{
          debuggerManager<-updateInbox(actorId, msg, false)@Debug;
          super^schedule(rcv,msg);
        } else: {
           if: isInStepReturn() then: {
             debuggerManager<-updateInbox(actorId, msg, false)@Debug;
             installFutureBreakpoint(msg);
             super^schedule(rcv,msg);
           } else: {
             pauseAndBuffer(rcv, msg, pausedState);
           }
       };
      } else: {
        def [isBreakpointed,bkpt] := isBreakpointed('receiverBreakpoints,rcv,msg);
        if: isBreakpointed then:{
          if: ( is: msg taggedAs: PauseResolve ) then:{
             installFutureBreakpoint(msg);
             super^schedule(rcv,msg);
          } else:{
            pauseAndBuffer(rcv, extendWithPauseBhv(msg,actorId,debuggerManager),
                           BREAKPOINT);
          };
        } else: {
           super^schedule(rcv,msg);
        };
      };
    } else: {
       if: (is: msg taggedAs: ExternalMessage) then: {
         super^schedule(rcv,msg);
       } else:{
         pauseAndBuffer(rcv,msg);
       }
    };
  };
};
```

Figure 10.10: Local manager's implementation of the schedule meta method.

next turn. If the message was executed as a result of a step-return command, the debug state is changed to running, and the debugger actor is notified of its resumption.

#### 10.4.2.3 Instrumenting Message Sending

Finally, recall from Section 4.6.3 that the `send` method is called before a message is queued on a far reference to be sent to another actor. Listing 10.11 shows the behaviour of the `send` method in the local manager's implicit mirror. Similarly to `schedule`'s implementation, `send` first checks the message's annotations. If the message has a `@Debug` annotation, default semantics for `send` are applied as it represents a debug-level message sent to update the debugger manager. When an application-level message is sent, the local manager first extends it with the turn information, and then checks whether the message hits a breakpoint stored in the `senderBreakpoints` hashmap. If the message hits a breakpoint, or if the local manager was instrumented to do a step-

```
def send(rcv, message){
  def types := tagsOf: message;
  if: (nil != (types.find: { |type| type.isSubtypeOf(Debug) })) then: {
    if: (!disconnectedFromDebuggerManager) then: {
      super^send(rcv,message);
    }
  } else: {
    def turn := updateTurnHistory(turnId, message);
    def msg := extendWithTurnInformation(turnId, turn.getCause().selector, message);
    def [isBreakpointed,bkpt] := isBreakpointed(`senderBreakpoints,rcv,msg);
    if: (isBreakpointed.or:{isInStepInto()}) then: {
      def newMsg := msg;
      if: (nil != bkpt) then: {
        if: bkpt.onEntry() then: {
          newMsg := extendWithPauseBhv(msg, actorId, debuggerManager);
        } else:{
          newMsg := extendWithPauseBhv(msg, actorId, debuggerManager,PauseResolve);
        };
      } else:{
        newMsg := extendWithPauseBhv(msg, actorId, debuggerManager);
      };
      def result := super^send(rcv,newMsg);
      if: isInStepInto() then: {
        debuggerManager<-updateMessageSent(actorId, msg)@Debug;
      };
      result;
    } else:{
      super^send(rcv,msg);
    };
  };
};
```

debug message

application-level message

Figure 10.11: Local manager's implementation of the send meta method.

into command, the local manager again extends the message with a `@Pause` annotation before sending it by calling the `extendWithPauseBhv` method (explained later in Section 10.4.4).

### 10.4.3 Breakpoints

As discussed in Section 10.3.2, a breakpoint can be categorized according to three basic properties: role, designation and objective. In the implementation, we define the behaviour common to all breakpoints in a trait in order to promote its reusability. The trait is "mixed into" different breakpoints in order to implement the breakpoint catalog shown in Figure 10.5.

```
def TBreakpoint := isolate:{
  def condition;
  def breakpointId;
  def breakpointTypes;
  def init(cond, types, bId := /.at.support.debugger.util.generateRandomId()){
    [self.condition,self.breakpointId,self.breakpointTypes] := [cond,bId,types];
  };
  def getBreakpointId() {self.breakpointId};
  def onEntry() {true};
  def ==(otherBreakpoint) {
    self.getBreakpointId == otherBreakpoint.getBreakpointId
  };
  def matches(rcv,msg) { self.condition(rcv,msg) };
  def getBreakpointTypes() { self.breakpointTypes };
  def isTaggedAs(typeTag) {
    { |return|
      self.breakpointTypes.each: { |t|
        if: t.isSubtypeOf(typeTag) then: { return(true)}
      };
      false;
    }.escape();
  };
};
```

The `TBreakpoint` trait represents a breakpoint as an object storing an identifier, a condition and breakpoint type tags. The `breakpointId` uniquely identifies the breakpoint and is used for storing the breakpoint in the debugger data structures. The `condition` variable stores an AmbientTalk closure encoding a boolean predicate that needs to be satisfied in order for a message to hit the breakpoint. The `breakpointTypes` denotes the role of the breakpoint. A breakpoint may be tagged with the `SenderBreakpoint` and/or `ReceiverBreakpoint` type tag. Finally, the `onEntry` method defines the objective of the breakpoint. By default all breakpoints are on-entry breakpoints but this method can be overridden by the composite object.

Note that the `TBreakpoint` trait expresses all REME-D's breakpoints as conditional breakpoints. Hence, code breakpoints have been built into REME-D in terms of conditional breakpoints as shown in the following listing:

```
def codeBreakpoint := isolate: {
  import /.at.support.debugger.util.TBreakpoint alias init := initBreakpoint;
  def filename;
  def lineNumber;
  def init(name, number, cond, types){
    def breakpointId := (name + "-" + number);
    self.initBreakpoint(cond, types, breakpointId);
    filename := name; lineNumber := number;
  };
  def getFilename() {filename};
  def getLinenumber() {lineNumber};
};
```

When a user places a breakpoint on a line of code on the UI, the debugger manager creates an instance of the corresponding code breakpoint and then, it passes it as argument in a `setBreakpoint` message to the necessary local manager(s). As such, a code breakpoint is declared to be an isolate (i.e., an object passed by copy). Listing 10.1 shows the implementation of all code-based breakpoints in the breakpoint catalog offered the user, namely message, message resolution and method breakpoints. They have been implemented as an extension to the `codeBreakpoint` prototype. The **`script:carrying:`** helper function allows the creation of a closure which is passed by copy in inter-actor messages copying into the closure scope the variables defined in the table given as argument. In the remainder of this section, we explain how the debugger handles breakpoints by describing message-type breakpoints.

### 10.4.3.1 Message Breakpoints

As shown in Listing 10.1 (lines 18-23), a message resolution breakpoint extends a message breakpoint prototype overriding the behaviour of the `onEntry` method. The `onEntry` method returns false, denoting an on-exit breakpoint. Since a message resolution breakpoint is a subtype of message breakpoint, the local manager will add it to its `senderBreakpoint` hashmap which is checked each time a message is sent. This happens when the `send` meta method calls the `isBreakpointed` helper function (cf. Figure 10.11, line 10). `isBreakpointed` returns a boolean if the outgoing message hits a sender breakpoint, and the matching breakpoint if any. If a matching breakpoint

Listing 10.1: Implementation of REME-D's code-based breakpoints

```
def messageBreakpoint := extend: codeBreakpoint with: {
  def init(name, number) {
    def cond := script: {|rcv, msg|
      def utilModule := /.at.support.debugger.util;
      def lineNumber := msg.getLocationLine();
      def filename := msg.getLocationFilename();
      def res := false;
      if: ((nil != lineNumber).and: { nil != filename }) then:{
        res :=  ((name == filename).and:{number == lineNumber})
      };
      res
    } carrying: `[name, number];
    def types := [utilModule.SenderBreakpoint, utilModule.MessageBreakpoint];
    super^init(name, number, cond, types);
  };
};
def messageResolvedBreakpoint := extend: messageBreakpoint with: {
  def init(name, number) {
    super^init(name, number);
    self.breakpointTypes := super^getBreakpointTypes()+[MessageResolveBreakpoint];
  };
  def onEntry() {false};
};
def methodBreakpoint := extend: codeBreakpoint with:{
  def init(name, number) {
    def cond := script: { |rcv, msg|
      def res := false;
      if: ((reflect: rcv).respondsTo(msg.selector)) then: {
        def method :=  (reflect: rcv).grabMethod(msg.selector);
        def sourceLocation := /.at.support.util.getSourceLocation(method);
        if: (nil != sourceLocation) then: {
          if: ((name == sourceLocation.fileName).and:{
            number == sourceLocation.line}) then: {  res := true }
        }
      };
      res
    } carrying: `[name, number];
    def types := [/.at.support.debugger.util.ReceiverBreakpoint];
    super^init(name, number, cond, types);
  };
};
```

is an on-entry breakpoint, then the debugger extends the message with a `@Pause` annotation before sending it. If a matching breakpoint is an on-exit breakpoint, then the debugger extends the message with a `@PauseResolve` annotation before sending it.

At a later point in time, when the receiver actor schedules the message, its local manager also checks whether it matches a breakpoint as shown in Figure 10.10 (line 20). The `isBreakpointed` helper function always returns true for messages annotated with the `@Pause` or `@PauseResolve` annotation. If the message is annotated with the `@Pause` annotation, the actor's execution is only paused. If the message is annotated with the `@PauseResolve` annotation, the local manager installs a new breakpoint called a *future resolution breakpoint* as shown in Figure 10.10 (line 23). The following code snippet shows the implementation of this breakpoint.

```
def futureResolutionBreakpoint := extend: conditionalBreakpoint with: {
  def init(msg) {
    def condition := {|receiver, message|
      def selector := messag.selector;
      (receiver == (reflect: msg).invokeField(msg, `future)).and:{
      (selector ==`resolveWithValue).or:{selector == `ruinWithException}
    }};
    super^init(condition, [/.at.support.debugger.util.SenderBreakpoint]);
  }};
```

A future resolution breakpoint is a conditional sender breakpoint whose goal is to hit the message carrying the return value of the original message computation. When the `resolveWithValue` or `ruinWithException` message carrying the result of the computation is sent back to the sender actor, the future resolution breakpoint will be hit, causing the message carrying the result to be annotated with a `@Pause` annotation.

The local manager also creates a future resolution breakpoint when executing a step-return command as shown in Figure 10.10 (lines 11-15). A similar implementation technique has also been used to implement a step-until command. However, a step-until command makes use of a conditional sender breakpoint created from the conditions the user introduces via REME-D's UI.

### 10.4.4 Infecting AmbientTalk VMs

As explained before an actor becomes infected when it receives a breakpointed message (a message which has hit a breakpoint). A breakpointed message is annotated by a `Pause` type or one of its subtypes (e.g., `PauseResolve` type) to pause the receiving actor execution. Listing 10.2 shows the implementation of breakpointed messages. The `extendWithPauseBhv` function takes four parameters: the message to breakpoint, and the type tag to annotate the message with, the identifier of the actor sending the message, and a reference to the debugger manager. The function returns a new message extending the original message encoding the infection of the actor receiving the message by overriding the method responsible for processing the message, called `process`. To this end, the `process` method (lines 8-21) installs a new local manager on the receiver actor (if necessary) by means of the `enableLocalManager` function. Recall from Section 4.6.2 that implicit mirrors on actors can be dynamically installed on an existing actor in order to override an actor's meta methods. The only require-

Listing 10.2: Implementation of breakpointed messages

```
 1  def extendWithPauseBhv(msg, actorId, debuggerManager, pauseType := Pause) {
 2    if: (is: msg taggedAs: pauseType ) then:{
 3      msg; // do not wrap it again.
 4    } else:{
 5      extend: msg with: { |actorId, debuggerManager|
 6        def lmModule := /.at.support.debugger.localManager;
 7        def alreadyPaused := false;
 8        def process(rcv) {
 9          def actor := reflectOnActor();
10          if: !(is: actor taggedAs: lmModule.LocalManagerModule) then: {
11            if: ((reflect: actor).respondsTo(`debuggable)) then:{
12              lmModule.enableLocalManager(true, false, debuggerManager);
13              rcv <+ self;
14            } else:{
15              raise: lmModule.XDebuggerException.new(
16                              "cannot infect an non-debuggable actor");
17            };
18          } else: { //local manager already enabled
19            super^process(rcv);
20          };
21        };
22        def getLocationLine(){super^getLocationLine() };
23        def getLocationFilename(){super^getLocationFilename()};
24        def getSenderActorId(){actorId};
25      } taggedAs: [pauseType, /.at.lang.types.Isolate];
26    };
27  };
```

ment for infecting an actor is that the receiving actor knows the source code for the local manager. The source code is included in the default AmbientTalk standard library, and thus accessible to any created actor (via the `/.at.support.debugger` namespace). Nevertheless, REME-D's implementation only infects the receiver actor if its actor mirror has a `debuggable` field. This field exists on those actors created within an AmbientTalk VM launched with the `-Xdebug` option.

We conclude our implementation overview by explaining how the `enableLocalManager` function works. As shown in Listing 10.3 (line 6), it installs a local manager by replacing the current actor mirror with a local manager's implicit actor mirror (returned from calling `makeLocalManager` from Figure 10.9). `enableLocalManager` is called to install a local manager either upon infection (as just shown in Listing 10.2 line 12), or when an actor is created as a result of evaluating code in a debugging session. It is also called to uninstall a local manager when a debugger actor leaves the debugging session (because of disconnection or as a result of a stop command).

Listing 10.3: Enabling a local manager's implicit mirror

```
 1  def enableLocalManager(enable, debuggingSession := true, debuggerManager := nil) {
 2    def actor := reflectOnActor();
 3    if: (enable) then: {
 4      if: (!(is: actor taggedAs: LocalManagerModule)) then: {
 5        def newProtocol := makeLocalManager(actor,debuggingSession,debuggerManager);
 6        actor.becomeMirroredBy: newProtocol;
 7      } else: { actor };
 8    } else:{
 9      actor.becomeMirroredBy: defaultActorMirror;
10    }
11  };
```

### 10.4.5 Implementation Status

The implementation discussed in this section is a "proof of concept" implementation the aim of which is to show that it is feasible to implement AODs using a mirror-based reflective architecture with the enhancements of Chapter 5. As such, it should not be considered as a stable, optimized, scalable debugger. Rather, it is a research artifact that serves as proof by construction that an AOD is viable.

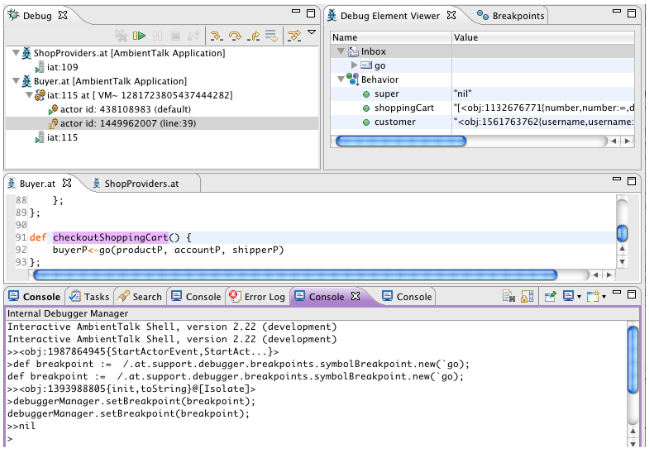
As previously mentioned, REME-D has been integrated with the AmbientTalk IDE for Eclipse as the debugger module. Nevertheless, REME-D can run independently of this particular front end. There exists another front end written in Java Swing developed within the context of Astudillo's master thesis [Ast12][4].

There are a number of features which are not currently integrated in the REME-D's UI, i.e., they do not have a dedicated UI component in the Eclipse IDE:

- causal link browsing.
- setting symbol, conditional and message resolution breakpoints.
- simulating network disconnections.

[4] Screenshots of the Java Swing front end are available at `http://patricio-astudillo-thesis.blogspot.com/`

In order for users to make use of those features, the Eclipse plugin has been extended with a console connected to the AmbientTalk VM running the debugger manager, called *AT Debugger Manager*. The user needs to instrument the debugger by explicitly invoking methods of the debugger manager from the command line in the "AT Debugger Manager" console. Figure 10.12 shows how a user can make use of such an interactive console to install a breakpoint on the `go` symbol. As shown, the result of such a symbol breakpoint pauses the buyer actor before executing the `go` method (analogously to the code breakpoint shown in figure 10.3).



```
ShopProviders.at [AmbientTalk Application]
    iat:109
Buyer.at [AmbientTalk Application]
    iat:115 at [ VM~ 1281723805437444282]
        actor id: 438108983 (default)
        actor id: 1449962007 (line:39)
    iat:115
```

| Name | Value |
|---|---|
| Inbox | |
| go | |
| Behavior | |
| super | "nil" |
| shoppingCart | "[<obj:1132676771{number,number:=,d |
| customer | "<obj:1561763762{username,username: |

```
    };
};

def checkoutShoppingCart() {
    buyerP<-go(productP, accountP, shipperP)
};
```

```
Internal Debugger Manager
Interactive AmbientTalk Shell, version 2.22 (development)
Interactive AmbientTalk Shell, version 2.22 (development)
>><obj:1987864945{StartActorEvent,StartAct...}>
>def breakpoint :=  /.at.support.debugger.breakpoints.symbolBreakpoint.new(`go);
def breakpoint :=  /.at.support.debugger.breakpoints.symbolBreakpoint.new(`go);
>><obj:1393988805{init,toString}@[Isolate]>
>debuggerManager.setBreakpoint(breakpoint);
debuggerManager.setBreakpoint(breakpoint);
>>nil
>
```

Figure 10.12: Eclipse plugin showing the AT Debugger Manager console.

It is important to mention that the AmbientTalk IDE for Eclipse actually contains the AmbientTalk/2 interpreter rather than the AmbientTalk/M dialect described in Chapter 5. As a result, there are a number of features which are not supported by REME-D when invoked from within AmbientTalk's Eclipse IDE:

**Lazy Introspection of Objects** When an actor is paused, the local manager is passed the actor's state in the `actorPaused` message including the object behaviour fields and its values. In order to minimize the amount of state to be transferred, REME-D's implementation in AmbientTalk/M wraps the values in a special kind of lazy reference (cf. Section 5.2.4) that works as follows. The debugger manager first obtains a far reference to the value, and only when the user unfolds the object's field, is the actual value passed to the UI in a `fieldUpdate` event.

**Push-based Introspection of Objects** REME-D's implementation in AmbientTalk/2 uses a *pull* approach to introspect objects. The object's fields and values are only requested to the corresponding local manager when a user unfolds an object in the state inspector. REME-D'S implementation in AmbientTalk/M combines this with a *push* approach using the method invocation protocol described in Section 5.3.1.2. The local manager places an after-observer whenever an assignment changes the value of the object's field being introspected (i.e., an object which is unfolded in the state inspector) to notify the UI of the new assigned value. As a result, REME-D conveniently updates the UI's values when necessary, avoiding that the user has to manually force a UI refresh by folding and unfolding objects in the state inspector.

**Network Failure Simulation** REME-D's implementation makes use of the reference management protocol described in Section 5.3.1.1 in order to simulate network failures. When a user wants to simulate the network disconnection of an actor, the debugger does not make use of the `network.offline` primitive because that would also disconnect the debugger actor itself. Rather, it iterates over all receptors created in the actor, and disconnects all of them except for the one to the local manager actor mirror.

In order for users to make use of the aforementioned features, they need to download the AmbientTalk/M interpreter and library projects and create their own distribution[5]. The AmbientTalk Eclipse IDE allows a user to use his own AmbientTalk distribution by specifying it in a custom debug configuration for the AmbientTalk file to debug.

## 10.5 Limitations and Future Work

There are a number of features which could be interesting to provide in an ambient-oriented debugger. First of all, since the focus of AOD is on inter-process communication, debugging sequential messages is currently not supported in REME-D . Including support for such messages would require adaptations to AmbientTalk's interpreter, so that the continuation stack is reified in the reflective layer. Second, the happened-before relationship provided when browsing causal links informs users of *all* possible places that may have caused a bug [SCM09]. However, it does not guide users about where to look first, nor to pinpoint when a bug occurred. In order to support developers in this task, we would like to combine features from omniscient debuggers [Lew03, PTP07]. For example, turn history could also be extended to include actor state history to be able to know in which context a particular field or variable was given a certain value. In order to provide stepping both forward an backwards in time, REME-D's implementation should modularize the stepping process (currently highly interwoven with the `schedule` meta method) into an abstraction similar to breakpoints. As a result, the `schedule` meta method would delegate the stepping functionality into the step abstraction passing by parameter the data structure "where to step". With respect to breakpoints, we believe that our breakpoint abstraction can be reused to place breakpoints on the message history, but it remains to be actually carried out.

In this work, we have not considered to support several debugging sessions at the same time. We assume that a device launching an application in debug mode hosts the debugger manager and that the rest of devices required for a debugging session would be dynamically added to the existing debugging session. It would be interesting to allow devices to participate in multiple debugging sessions. For example, this can be beneficial for building a omniscient debugger in the face of partial history information. A debugger manager could communicate with other active debugger managers in the network to obtain information about an actor which has left a session.

At a technical level, REME-D 's implementation available in the Eclipse IDE still suffers from a number of limitations which was also confirmed by our user study that we discuss in the next chapter. Amongst them, we plan to improve the actor view and introduce a dedicated message view to facilitate the browsing of message history. In fact, the ultimate goal is to re-implement the Eclipse UI without using the Debug

[5] Instructions how to check-out those projects, and create a distribution are available at `http://tinyurl.com/6wak2p7`

Framework. Many of the UI problems experienced stems from the fact that the Debug Framework has been designed for Java programs. Since AmbientTalk is a distributed dynamic language, we need to tweak the UI implementation for supporting certain features, e.g., debugging sessions are not meant to be open in Eclipse. We would also like to develop a dedicated UI for the Android platform in order to explore “live debugging” of applications running on Android devices.

Finally, the most relevant limitation that we plan to tackle in future work is the debugging of custom referencing abstractions including leased references. Although the current implementation of REME-D makes use of the reflective architecture introduced in Chapter 5, it lacks a layered design. As a result, it would have a significant impact on the debugger architecture to be able to debug other reflective language constructs such as leased references while avoiding interference. First steps towards a layered design have happened during decoupling the Eclipse UI from the debugger core, which allows us to write unit tests for the debugger. Moreover, since interactions within the debugger are stratified by means of the `@Debug` annotation, the architecture has the basic blocks to be able to “debug the debugger”. However, it remains to be investigated how we can debug other reflective language constructs in an easy and modular way without introducing interference while minimizing the debugger’s probe effect. More importantly in the case of leased references, it remains to be investigated how and what it means to “debug a time changing value”, e.g., should we pause all the lease terms when an actor’s execution is stopped or not?, should we simulate the passage of time?, etc.

## 10.6 Conclusion

In this chapter, we have explored tool support in the face of partial failures in the form of a debugger. In ambient-oriented programming, the complexity of programming in a distributed setting is married with the network fragility and open topology of mobile applications. To address the challenges faced when debugging ambient-oriented applications, we introduced an online ambient-oriented debugger called REME-D. REME-D’s principal contribution is that it implements the features of ambient-oriented debuggers as an ambient-oriented application which incorporates breakpoint-based debugging methodology where the focus is placed on the exchange of asynchronous messages between actors. More concretely, REME-D adapts features from breakpoint-based debuggers to event loop concurrency —actor state inspection, message breakpoints, stepping over or into turns— , while incorporating for online usage features from post-mortem, message-oriented debuggers —browsing causal links. To respond to the openness of MANETs, REME-D proposes epidemic debugging: it can install itself on newly discovered devices, a process akin to an infection in which REME-D spreads to devices joining the debugging session. Devices can leave the debugging session, either due to communication failures or in response to a user action, without disrupting the debugging of the remaining participants. REME-D implements those features by exploiting the reflective API described in Chapter 5, resulting in a modular, reusable and flexible design that shows that it is possible to build tool support in tandem with the programming support for dealing with partial failures.

# Chapter 11

# Pre-experimental User Study for REME-D

In order to assess the feasibility of an ambient-oriented debugger, we tested REME-D in a study with 22 subjects which follows a *one-group pretest-posttest pre-experimental* design. The subjects were asked to use REME-D's implementation available in AmbientTalk's Eclipse IDE during an assignment the goal of which was to understand and debug the shopping application used as the running example in the previous chapter (cf. Section 10.1.1). In this chapter, we describe the conducted experiment and its results.

## 11.1 Quasi-Experiments

The goal of our experiment was to test our debugger on a number of points in order to study the practical use of its features. There are three conditions to take into account when choosing the most appropriate empirical study for this experiment. First, we are testing a new tool. Second, the AmbientTalk user base is relatively small (estimated to 8 active researchers, and around 100 occasional programmers including master students and outside contributors). Third, we dispose of a relatively short amount of time (which does not allow us to study participants over a long period of time). Given those restrictions, the most reliable and feasible type of experiment is a *quasi-experiment* [CS63]. A quasi-experiment is an empirical study which lacks random assignment on selection of the groups or other factors being studied. Moore describes in [A.08] a number of situations in which it is appropriate to conduct a quasi-experiment study instead of a random assignment evaluation. Our experiment certainly complies with at least three of the described situations: (1) the pool of potential participants is too small to fill both a treatment and a control group, (2) it is impossible to avoid "contamination" of the control group[1], and (3) REME-D was still under development at the time of the study.

Note that a quasi-experiment, as opposed to a (randomized) scientific experiment, does not allow us to make any founded claim regarding the usability of REME-D. However, it does provide insights about how real users perceive and value the features of an ambient-oriented debugger. Still, quasi-experiments are an established practice

[1] Contamination may happen e.g., when students in the control group may talk about the study during lunch or breaks from classes, making the control group be influenced by the study.

in software engineering [KDHKS09][2], and have been previously applied for providing an initial assessment of software tools integrated in Eclipse [DWZVD09].

## 11.2 Study Design

We employed a *one-group pretest-posttest quasi-experiment design* [CS63] in our user study. This experiment consists of only one group of 22 participants which is subject to a test before the experiment is conducted (called a *pretest*). The pretest serves as the baseline to quantify the expectations of a participant regarding an ambient-oriented debugger before being introduced to REME-D. After having used REME-D in a number of debugging tasks, participants are asked to fill in a test (called a *posttest*) that measures their perception of the tool. Pretest and posttest usually employ the same questions varying the independent variable, i.e., introducing REME-D. By comparing the pretest and posttest results, we can measure how exposure to REME-D influenced the perception of ambient-oriented debugging, and which features of REME-D were deemed useful by participants.

The questionnaires used for the pretest and posttest employ close-end matrix questions in which participants need to rate a number of statements on a five-point Likert scale, i.e., a 1-5 scale ranging from “totally disagree” to “totally agree”. Likert scales are easy for respondents to fill in and provide enough freedom to express their opinion. However, they may be subject to distortion as respondents may start agreeing with statements as presented, or following a certain pattern instead of expressing their opinion (*acquiescence bias*). In order to avoid this bias, we intermingle the number of consecutive positive and negative statements so that respondents have more difficulties to find out trivial patterns.

In the experiment, participants were invited for a session that took between 45 to 75 minutes in total. At the start of the session, the participants were asked to fill in the pretest. Afterwards, they received a short demonstration (that lasted for about 15 to 20 minutes) of the features of an ambient-oriented debugger in REME-D. Following this demonstration, the participants were asked to use REME-D to identify and solve bugs in the shopping application (cf. Section 10.1.1). After the participants completed their assignment, or reached the end of the predefined amount of time, they had to fill in a second questionnaire serving as a posttest.

### 11.2.1 Pretest design

The pretest measured the expectations prior to using REME-D. It consists of 22 statements structured along four themes. Each theme aims to determine possible external variables that might influence the dependent variables.

**Personal background** We ask some personal details including age, education level, and the top 3 programming languages they were most comfortable with.

**Development experience** A number of statements were included to get information about the participant’s experience in developing software including experience with AmbientTalk and the Eclipse IDE.

[2] Kampenes et al. studied in [KDHKS09] 113 experiments reported on nine major software engineering journals between 1993-2002, and detected that 35% of them were quasi-experiments.

**Attitude towards debugging** In order to assess the participant's standpoint towards debugging, a number of statements were included that relate to debugging.

**Expectations from an ambient-oriented debugger like REME-D** Finally, a number of statements were included to measure the actual dependent variable, namely REME-D . In particular, they assess the participant's expectations w.r.t an ambient-oriented debugger and the debugging features to be supported.

We left enough space for participants to write down some comments about desired features in an ambient-oriented debugger, and features of the AmbientTalk language that are perceived to make the developing of AmbientTalk applications hard.

The exact list of statements used in the pretest can be found in Appendix B.1.

### 11.2.2 Posttest design

The main goal of the posttest is to measure whether the participant's expectations with regard to an ambient-oriented debugger have been fulfilled, and whether REME-D is considered to be a useful debugger. The posttest consists of 24 statements structured according to a number of issues:

**Assignments Experience.** The first series of statements aims to assess interference by external variables. In particular, they relate to the participant's experience with the assignment. Was the assignment too hard? Did the participant find the assignment representative for the kind of bugs he previously encountered while developing software in AmbientTalk? Did the participant feel time pressure that may have influenced his performance or perception? Was the assignment sufficiently interesting?

**Value of an ambient-oriented debugger.** To get an impression of the participant's perception of an instance of an ambient-oriented debugger, a number of statements were included in relation to REME-D. These statements aim to assess whether REME-D provides a better or quicker methodology in the process of debugging and understanding AmbientTalk applications. Does an ambient-oriented debugger provide any added value? Does REME-D allow developers to debug AmbientTalk applications more efficiently? Does REME-D allow developers to understand AmbientTalk applications more effectively? Does REME-D allow developers to solve realistic bugs?

**UI Experience.** A number of statements were included to assess the participant's experience with the REME-D user interface in the Eclipse IDE. The goal of these statements is twofold. First, they aim to gauge to what extent the participant noticed the dedicated UI components for REME-D available in Eclipse, and actively used them. Were the debugging features clear and easy to find in the UI? Does the debug element view provide a good overview of the actor's state? Secondly, they aim to assess interference that might influence the perception of REME-D's features. Was the UI easy to use? Does the debugger require a better user interface?

**Value of REME-D features.** To get a better impression of the participant's perception of the debugging features, we added two kinds of questions related to each debugging feature introduced in the tool demonstration, namely message breakpoints, step-into, step-over and pause command, infection of devices. First, participants were asked to indicate how often they used REME-D's features. Second, the usefulness of each of the REME-D features was asked for. If the feature was never used, the participants were asked to indicate how useful they thought the feature would be.

We also left enough space for participants to write down some comments or suggestions they had about the tool and the assignment.

The exact list of statements used in the pretest can be found in Appendix B.2.

### 11.2.3 Debugging Assignment

In the experiment, the participant was asked to complete a number of tasks relative to the debugging process of an AmbientTalk application, namely, the shopping application. When choosing an application for the study several requirements were taken into account. First, the application should not be too complex so that the assignment could be finalized within the time frame. However, the application should be representative of a real world ambient-oriented application. Finally, the experiment setup for the target application should exclude as many external variables as possible. The shopping application scenario satisfies these requirements. It is a simple yet representative application that has been previously used as running scenario of a distributed debugger [SCM09]. In addition, it can be easily adapted to run in two AmbientTalk virtual machines, so that participants do not need to spend too much time on the experimental setup.

The study involved a debugging assignment which consisted of two tasks for which we introduced errors in the shopping application. The assignment first described the shopping checkout protocol shown in Figure 10.1, and detailed what was the expected behaviour when running it. In the first task, the participants were asked to use REME-D to find out why the shopping checkout protocol did not work properly and fix the problem. The second task described an extension to the shopping checkout protocol in which the buyer contacts a warranty broker to propose a warranty for the purchases item to the client. Since the quota returned was always negative, participants were asked to investigate the protocol and determine what the problem was.

Appendix B.3 details the final debugging assignment used in the experiment. In addition to the assignment text, participants were also given introductory documentation on REME-D explained during the tool demonstration. Finally, the necessary code for the experiment was made available as an Eclipse Java project including two AmbientTalk files (one for the shopping application, and a second one including the code for the warranty broker).[3]

## 11.3 Participants profile

In the study we observed 22 participants working on a number of debugging tasks. The participants were recruited from within the computer science department of our university, in particular, they were all enrolled in a master or PhD program. More concretely, 13 participants hold a bachelor degree and were close to obtaining a MSc degree, and the remaining 9 hold a MSc degree and were (relatively) close to a PhD degree. All

[3] All the material offered to the participants is available at `http://tinyurl.com/debuggingSessionMaterial`.

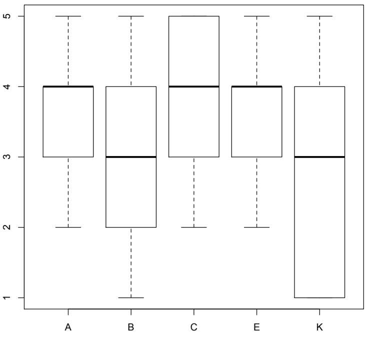


Figure 11.1: Boxplot of the experience of the participants: (A) development experience (B) distributed development experience (C) understanding of AmOP in AmbientTalk (E) Eclipse experience (K) online debuggers experience.

| | Top 1 | Top 2 | Top 3 |
|---|---|---|---|
| Scheme | 11 | 1 | 2 |
| Java | 3 | 11 | 6 |
| Ruby | 3 | 1 | 2 |
| C++ | 2 | 1 | 0 |
| C# | 1 | 1 | 0 |
| Haskell | 1 | 1 | 0 |
| AmbientTalk | 1 | 0 | 2 |
| SmallTalk | 0 | 0 | 6 |
| Python | 0 | 2 | 0 |
| C | 0 | 1 | 2 |
| Lisp | 0 | 1 | 0 |
| Perl | 0 | 1 | 0 |
| JavaScript | 0 | 1 | 0 |
| Objective C | 0 | 1 | 0 |
| ASP.NET | 0 | 0 | 1 |
| Total | 22 | 22 | 21 |

Figure 11.2: The participants' expertise with software languages.

participants in the study were required to have experience with AmbientTalk. Finally, all participants were of age 21 to 27.

During the pretest we inquired about their knowledge of development in general, distributed applications development, ambient-oriented programming in AmbientTalk, the use of Eclipse and online debuggers. Figure 11.1 provides a summary of this inquiry in a boxplot. Most participants considered themselves rather experienced software developers (mean score 4), but not particularly experienced in developing distributed applications (mean score 3). However, all the participants understand the principles of ambient-oriented programming in AmbientTalk (C). As for their knowledge of software platforms and tools, all the participants have indicated to be rather familiar with Eclipse (mean score 4), but most participants consider themselves to only have limited experience with online debuggers.

In order to get a better impression of the participant's developer profile, participants were also asked to provide their expertise with particular technologies, i.e., the three programming languages in which they would consider themselves to be most proficient. The results are shown in Table 11.2. All the participants are experienced in a dynamic language like AmbientTalk, and are knowledgeable of Java. None of the participants had prior knowledge of REME-D.

Finally, the pretest also included a number of questions to measure the participants' attitude towards AmbientTalk, Eclipse IDE and debugging. Table B.1 (in Appendix B.1) includes the exact statements for those questions, and Figure 11.3 shows the answers to these questions in a boxplot. As shown in Table B.1, several questions were asked relative to debugging. Next to the boxplot, Figure 11.4 provides a summary of the participants' attitude towards debugging in a radar diagram; each branch represents a single question; the bold line shows the average and the colored surface indicates the range of given answers (calculated as the average ± the standard deviation).

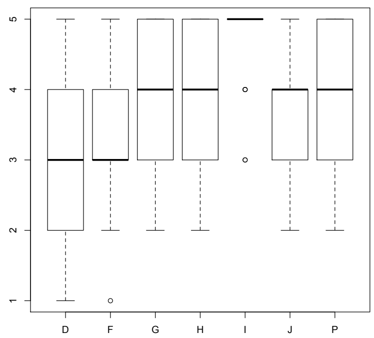


Figure 11.3: Boxplot of the participants' attitude towards (D) AmbientTalk, (F) Eclipse IDE, and (G-J,P) debugging.

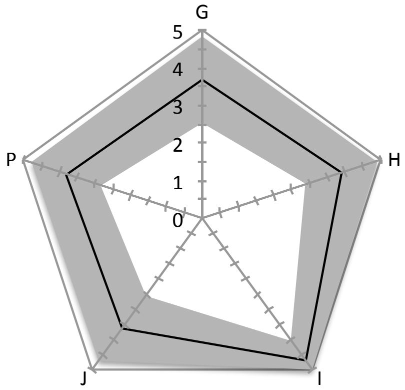


Figure 11.4: Radar diagram of the participants' attitude towards debugging: (G) development tools can prevent a lot of bugs (H) debugging distributed programs is hard (I) debuggers are a helpful tool to find errors in programs (J) debuggers are a helpful tool to understand programs (P) a debugger for AmbientTalk is needed.

The participants' attitude towards debugging is pretty consistent along the different questions. With the exception of two respondents, all the participants strongly agree that debuggers are a helpful tool to find bugs in programs (I). In addition, they generally seem to agree that debuggers are a helpful tool to understand programs (J). With only three ratings lower than 3, participants acknowledge that debugging distributed programs is hard (H). Most of the participants also agree that better tools can prevent bugs (G, mean scores 4). More importantly, with only one rating lower than 3, participants are mostly unanimous about the need for a debugger for AmbientTalk (P).

As for their attitude to the software technologies and tools relevant to the study, most participants find the Eclipse IDE suitable for developing distributed applications in AmbientTalk (F) having only two participants rating the statement lower than 3. Unfortunately, the results on the participants' attitude towards AmbientTalk are not conclusive (D).

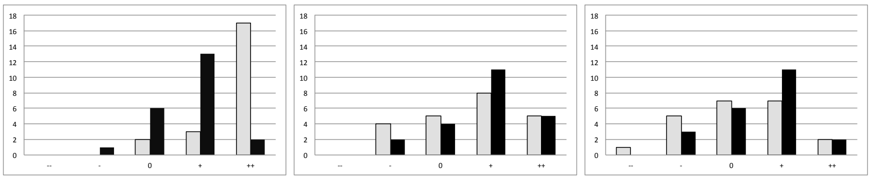
## 11.4 Results

In this section, we discuss the main results from the experiment[4].

### 11.4.1 Pretest-Posttest

We discuss the assignment experience and the way exposure to REME-D influenced the participants's perception of debugging ambient-oriented applications in AmbientTalk by comparing the results obtained from the pretest and posttest. Figure 11.5 provides an overview of the results. Overall, REME-D was well-received by the participants. As

[4] All the raw data including all 22 filled pre/posttests questionnaires is available at `http://code.google.com/p/ambienttalk/downloads`



(a) Value as a tool to find bugs in programs (b) Value as a tool to understand programs (c) Value as a tool to ease programming in AmbientTalk

Figure 11.5: Comparison of the participants' expectations (pretest depicted in grey) and experiences (posttest depicted in black) after using REME-D ; X axis depicts the 5-point Likert scale, and Y axis is the number of participants that selected each point.

can be seen in Figure 11.5 (a), most participants were positive with respect to the value of REME-D as a tool to help them find bugs in their programs with the exception of one participant. However, we can observe that their answers are more spread compared to the pretest, in which most participants strongly agree with the statement. Hence, REME-D did not meet the participants expectations in this regard. In our discussion with the participants after the session, they expressed their doubts about the suitability of the assignment, as some of them did not seem convinced that the types of bugs included are representative of real bugs. We will further discuss this point in the section about threats to validity.

Regarding the usefulness of REME-D as a program understanding tool, all participants indicate that the tool helps them to understand AmbientTalk programs (Figure 11.5 (b)). Actually, we can observe that the perception of the value of a debugger as a software understanding tool has improved in the posttest after working with REME-D. In the pretest, more participants expressed their reservations regarding a debugger as a tool helpful to understand programs. The posttest revealed that participants had a more positive attitude towards the statement after exposure to the tool, reducing the number of participants giving lower scores.

As for the use of REME-D as a means to make ambient-oriented programming in AmbientTalk easier (Figure 11.5 (c)), results show that most participants are more positive in the posttest. It is interesting to remark that 4 out of 22 participants change their opinion on this point after working with REME-D. We regard this result as encouraging since all participants stated in the pretest that they understand the principles of ambient-oriented programming in AmbientTalk.

While these are preliminary findings, we think the results are positive. They show that REME-D is capable of improving program understanding for AmbientTalk applications, and eases their development. Participants also seem convinced that the tool can help reduce the time to debug distributed AmbientTalk programs when asked in the posttest (question H in posttest; cf. Table B.2). This inquiry's mean scores 4 in the posttest, having only 2 participants rating the statement lower than 3.

#### 11.4.1.1 Features of REME-D

The posttest also included questions regarding the usefulness of REME-D's features. Table B.2 (in Appendix B.2) includes the exact statements for those questions, and Figure 11.6 shows a summary of participants' evaluation of these features. In gen-

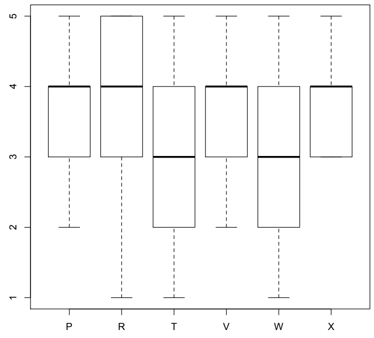


Figure 11.6: Boxplot of the participants' appreciation of the features of REME-D. (P) message breakpoints (R) step-into command (T) step-over command (V) pause actor command (W) control over program execution (X) infection of VMs.

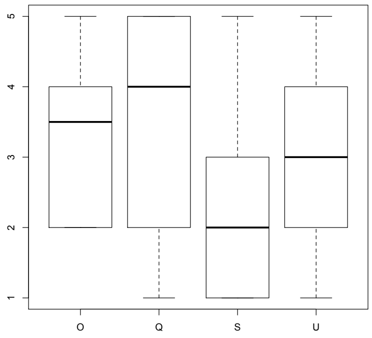


Figure 11.7: Boxplot of the participants' usage of the features of REME-D. (O) message breakpoints (Q) step-into command (S) step-over command (U) pause actor command.

eral, participants seem to have valued message breakpoints, step-into command, pause command and the infection of other VMs (all means scored 4). In particular, only one participant rated message breakpoints and pause command lower than 3. In addition, all participants were convinced of the usefulness of the infection feature; none of them rated this feature lower than 3. On the other hand, participants were rather neutral towards step-over command and the control over the execution of an AmbientTalk program. These results may be explained because the participants did use that features less or not at all. Figure 11.7 summarizes the participants's opinion about the frequency of usage of some of REME-D's features. Participants indeed did not use that many step-over commands during the assignment; only 4 participants score this statement higher than 3. Most participants often used message breakpoints, and the step-into command.

Regarding the effectiveness of the representation of REME-D's features in the Eclipse IDE, the opinions of participants in the posttest were reasonably positive. Figure 11.8 provides a summary of the participants' experience with REME-D's UI in a radar diagram. The formulation of each question can be found in Table B.2 (in Appendix B.2). Most participants did appreciate the actor and debug element views provided in the Eclipse IDE (mean for question L and M score 4). In addition, all participants find REME-D's UI easy to use with the exception of one participant who "strongly disagreed". That participant was totally disappointed with REME-D's UI since he is also the only participant who did not find debugging features clear and accessible in the UI and strongly disagree when asked about the usefulness of the actor and debug element views. Not surprisingly, he is one of the 4 participants that strongly believe that in essence REME-D is helpful, but it requires a better UI.

### 11.4.2 Observations

While working with REME-D, participants encountered a number of issues related to REME-D's UI. First, the debug element view was not correctly updated when an actor stopped due to a breakpoint. To overcome this issue, they were told to click

| Feature description | #comments |
|---|---|
| **Expected features supported in REME-D at the time of the study** | |
| inspecting actor state | 5 |
| inspecting mailbox contents | 4 |
| breakpoints | 3 |
| step-by-step execution | 2 |
| pause command | 1 |
| **Expected features supported in current version of REME-D** | |
| inspecting state inside of an object | 3 |
| simulation of disconnected scenarios | 2 |
| message history | 2 |
| breakpoints extensions | 2 |
| decoupling of debugger core functionality from Eclipse IDE | 1 |
| **Expected features interesting for future research in REME-D** | |
| mapping breakpointed messages to lines of code | 1 |
| better visualization of actor view | 1 |
| adding message view | 1 |

Table 11.1: Summary of comments about expected features.

on the actor label again so that Eclipse Debug Framework would trigger a UI refresh. Second, participants were confused by the way the UI presented the list of messages sent during a turn. The list was shown under the actor name in which they instrument the debugger to step into a message. However, when clicking on the message, the debug element view did not show the message state (i.e., the arguments state), raising a UI exception. This is because REME-D 's UI was not instrumented to show them that information. Participants were also slightly confused by the way the tool presents the actor participating in the debugging session in the actor view. Three participants explicitly included in the posttest comments to improve this view, e.g., “actors should get names instead of line numbers”.

Interestingly, more than half of participants (12 out of 22) left feedback on the pretest and posttest questionnaires. All the pretest comments are about expected features in the debugger. In contrast, the posttest comments mainly included improvements on the REME-D 's UI which relate to the above mentioned issues. In particular, 3 participants complained about the stepping functionality (e.g., “stepping-in was not very perfect. I got lost on where the current execution step was [..]”), and three more about the limitations of message state inspection.

Table 11.1 provides an overview of the expected features for a debugger for AmbientTalk applications that participants freely indicated in the section for comments. It maps each expected feature to the number of comments that explicitly refer to the feature. We have classified the table according to three kinds of features: features which were expected and supported in REME-D's version used during the study, features which we identified to be relevant, and have been incorporated in the version described in this chapter, and interesting features with which this work could be continued. It is interesting to remark that the most recurring expected features were already supported in REME-D at the time of the study. Note that although REME-D already provided some inspection of the actor state reachable from the behaviour, it was not possible to interactively inspect any kind of object (e.g. messages sent or in the inbox).

## 11.5 Threats to validity

As previously explained, a quasi-experiment study does not allow us to make any generalized claims regarding the usability of REME-D. Instead, quasi-experiments do allow us to observe how potential users perceived our tool. However, quasi-experiments are subject to concerns regarding the validity of the observations resulting from the experiment. According to the known guidelines for quasi-experimental research designs, in the next two subsections we discuss the threads to *internal validity*, and to *external validity* of our experiment. Internal validity considers the validity of cause-effect inferences made during the experiment, while external validity considers the validity of generalized inferences (or how wrong we are when making generalized observations from the study).

### 11.5.1 Internal Validity

The analysis of the experiment's outcomes assumes that REME-D is the only factor influencing the dependent variables. However, several factors may have potentially interfered in the participants' perception of REME-D.

First, participants may have felt inclined to answer positively to the tool. We mitigated this concern by making clear to participants that they did not have to please anybody, and that only honest answers were valuable.

Second, the introductory demonstration might have biased participants towards using the REME-D's features shown to them. To counter this effect, we explained all features of REME-D but only showed participants where they could find REME-D features in Eclipse, and the basic REME-D views. Participants were told they could use any feature they liked.

Third, the assignment executed by our participants might be too simplistic or hard. To assess this risk, the posttest included a set of questions to measure the participants' experience with the tasks performed in the assignment. Figure 11.9 provides a summary of the participants' experience with the assignment in a radar diagram. Recall that each branch represents a single question; the bold line shows the average and the colored surface indicates the range of given answers. The formulation of each question can be found in Table B.2 (in Appendix B.2). The results show that participants generally did not find the assignment too hard (question A) with the exception of two participants, and find the experiment interesting to do (question B).

Finally, the duration of the experiment may also have influenced the internal validity. We also included some questions for this inquiry. The results, shown in Figure 11.9, reveal that participants do not seem to have experienced time pressure (question D) and were satisfied with the help received to complete the assignment (question E).

### 11.5.2 External Validity

For the most part, the generalizability of the results depends on three major concerns [MZS$^+$10]: representativeness of the participants, suitability of the selected case, and the degree to which the bugs encountered in the assignment are representative of real-world bugs.

A risk exists concerning the composition of the group of participants: since all participants were computer science students or researchers, they might not form a representative sample of software developers. While all participants rated themselves as

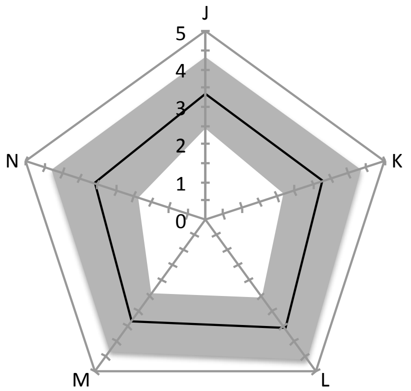


Figure 11.8: Radar diagram of the participants' experience with REME-D's UI in Eclipse: (K) REME-D's debugging features are clear and accessible in the UI (L) the Actor view gives a good overview of the state of the application (M) the Debug Element View gives a good overview of the state of an actor (N) REME-D is helpful but needs a better user interface.

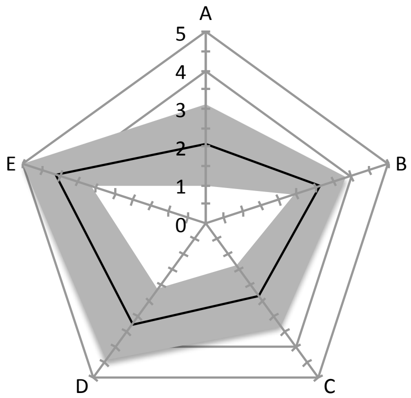


Figure 11.9: Radar diagram of the participants' experience with the assignment: (A) the assignment was too easy for me (B) the assignment was very interesting to do (C) the assignment represents the kind of bugs I have encountered in AmbientTalk (D) I would have liked more time to complete the assignment (E) I had enough help in completing the assignment.

expert software developers, they had different degrees of expertise with distributed software development, Eclipse, AmbientTalk and debugging (as discussed in Section 11.3). It is important to remark that the computer science master students (60% of our participants) had only been exposed to AmbientTalk for a month when the study took place. Although we admit that the learning curve involved may have impacted the results, participants were proficient in very similar languages (as discussed in Section 11.3). Hence, we think this impact is limited. In addition, participants were allowed to ask questions about the technology involved in the study at any time.

As previously explained, the shopping application is a "showcase" application. Although it has been used in previous research efforts in debugging [SCM09], the risk exists that the application is not representative of a real-world ambient-oriented application. Note that the assignment only tackled a part of the application (the checkout protocol) which was adapted so that participants could understand and debug it within the duration of the experiment.

Finally, the bugs encountered in the assignment may not have been representative of real-world bugs. Considering that we wanted to keep the amount of time necessary to execute the assignment manageable, the risk exists that the assignment did not capture the complexity associated with real-life bugs in ambient-oriented applications. Indeed, results show that participants did not find the assignment to be representative of the kind of bugs they encountered while developing in AmbientTalk (question C in Figure 11.9). This risk was also confirmed by some participants during discussions after the session.

## 11.6 Conclusion

In this chapter, we have described the setup and execution of a one-group pretest-posttest quasi-experiment study conducted for evaluating REME-D. The goal of this evaluation was to get an impression of the practical usability of the features proposed by an ambient-oriented debugger, rather than getting truly scientific results (which cannot be achieved with a quasi-experiment design because it lacks a random selection and assignment). Considering the results and observations described in the previous sections, the user study has provided us with three valuable insights. First, the features that participants actually expected from an ambient-oriented debugger were indeed supported in REME-D . This observation is based on the analysis of both the pretest statements and the suggestions that participants freely left on space provided for comments. Second, participants valued REME-D as a program understanding tool suited to make ambient-oriented programming in AmbientTalk easier. Finally, the Eclipse UI interface is relevant to how users perceive and value the features of REME-D , and it requires some attention.

# Part III

# Conclusion

# Chapter 12

# Conclusion and Future Work

This chapter concludes our study of a software development platform for the systematic construction of MANET applications that deals with *partial failures*. Before enumerating some interesting avenues for future research, we revisit the research goals with hindsight and we restate the contributions of the dissertation.

## 12.1 Research Goals

This dissertation studied programming language abstractions and tool support for MANET applications that incorporate concepts to deal with the effects caused by partial failures. In Section 1.3, we stated four research goals which we now review and discuss the extent to which they have been achieved.

- **We investigate programming language abstractions for failure handling in MANET applications.** In Chapter 3 we proposed a set of criteria for any adequate failure handling model to be used in a MANET. We employed these criteria in Chapter 3 to study the state of the art distributed programming languages and middleware. Our survey revealed that combining a decoupled communication model with leasing provides a solution for devising a failure handling model in MANETs. Based on this, we have investigated how to design a leasing model that operates in MANETs. To this end, Chapter 6 identified four criteria that a leasing model needs to fulfill to be used in a MANET, and proposed the notion of *ambient-oriented leasing*.

- **We explore whether these failure handling abstractions can be integrated in a distribution model which is not based on object-oriented programming**. Our survey of related work in Chapter 3 showed that data-driven models such as tuple spaces or publish/subscribe provide a loosely coupling of processes, making them suitable for a mobile environment. This led us to explore a first integration of ambient-oriented leasing in tuple spaces in Chapter 7. Ambient-oriented leasing has subsequently been integrated with the distributed object oriented programming language concepts of AmbientTalk.

- **We study tool support in the form a debugger in order to help programmers get a better understanding of the dynamic behaviour of a MANET application**. Our survey of state of the art on debugging tools in Chapter 9 showed that existing debuggers for distributed systems do not support debugging in the face of partial failures. This led us to design and study an *ambient-oriented debugger* in Chapter 10. It was also our explicit goal to investigate debugging support in the face of partial failures in tandem with the programming support for dealing with the effects of partial failures.

- Finally, **we investigate a novel distributed reflective architecture that enables the development of both programming language and debugging support for partial failures in MANET applications.** This led us to rethink the meta-level model of prior ambient-oriented languages in Chapter 5 and propose the *transmitter-receptor* meta-level model which reconciles mirror-based reflection with remote object references. Even though the transmitter-receptor model was technically added to our AmbientTalk language, we argue that it can be translated to any actor-based programming language such as E.

We achieved these goals following a "proof by construction" methodology. We have done so by developing novel programming language abstractions and tool support in an ambient-oriented programming language. The resulting artifacts validate the main concepts put forward in this dissertation:

**AmbientTalk/M** is an extension to the AmbientTalk/2 programming language. Its key feature in respect to our research goals is that it embodies the aforementioned transmitter-receptor model on which we conducted our experiments for failure handling language abstractions. Moreover, it incorporates an observer mechanism on mirrors that also helped us to implement tool support reflectively in the language itself. In short, the language provides an extensive meta-level architecture specially designed to ease the development of abstractions for distribution and failure handling in a MANET. Conversely, one might say that the language and tool experiments form a validation of the transmitter-receptor model.

**Leased Object References** are time-decoupled object references which deal with both transient and permanent failures by integrating the notion of leasing with object designation. Their most distinguishing features is that they integrate a number of useful leasing patterns while enabling meta-level engineers to build custom leased references encoding differing leasing semantics. Leased object references are the central abstraction of our leasing model for MANETs. We also integrated leasing into message passing by means of *due-type messages* that allow developers to deviate from the default message delivery guarantees provided by a leased reference. Finally, we studied various scoping abstractions to decrease the programming effort introduced by leased references and due-type messages. All together, they form an extensive set of abstractions for dealing with the effects engendered by partial failures.

**TOTAM** is a novel tuple space model with combines a replication-based tuple space model with ambient-oriented leasing enabling tuple management in the face of partial failures. The design and implementation of TOTAM demonstrates that the principles of ambient-oriented leasing are independent of the distributed object-oriented communication model and that they can be translated to a data-driven model such as tuple spaces.

**REME-D** is an ambient-oriented debugger that was developed to complement AmbientTalk programmer's toolbox with debugging support. More precisely, REME-

D is an online debugging tool that proposes novel facilities to deal with the hardware characteristics of MANETs (epidemic debugging, and support for partial failures) and which integrates techniques from traditional sequential debuggers (stepping and state inspection) and distributed debuggers (event-based debugging, message breakpoints) with ambient-oriented programming. The design and implementation of REME-D demonstrates that it is possible to build debugging support reflectively on top of an AmOP language.

## 12.2 Restating the Contributions

The following summarizes the contributions of each chapter with regard to the research goals:

- In Chapter 2, we formulated eight *criteria for an adequate failure handling model in MANETs* (cf. Table 2.1). We motivated each criterion based on hardware characteristics of MANETs. The criteria were organized according to the key indicators for the design of a failure handling model, namely, communication, state consistency and memory management. Decoupled communication (**C1**) is key for enabling communication over volatile connections, while a high-level representation of failures (**C2**) decouples low-level networking connectivity from a high-level application connection. Reacting to network connectivity (**C3**) is essential to support network-awareness at application and tool support level. With respect to state consistency, local failure recovery (**C4**) allows applications to perform failure handling without intervention of remote parties, and application-dependent failures handling strategies (**C5**) enables processes to define the most appropriate strategy to react to partial failures. From a memory management point of view, soundness should be relaxed (**C6**) so that data remains valid during intermittent disconnections, and contractual memory management (**C7**) enables reclamation of data in the face of permanent disconnections. Finally, in order to support the development of tool support, it is necessary incorporate support for triggering failure handling code explicitly (**C8**).

- In Chapter 3, we survey a number of representative distributed programming languages and middleware that satisfy one of more of these criteria. Table 3.1 evaluates each approach for the aforementioned criteria. Based on our analysis of related work, *we concluded that a failure handling model should be designed around the concept of leasing combined with a decoupled communication model.*

- In Chapter 5, we first discussed where AmbientTalk's meta-level architecture falls short to support the development of failure handling abstractions and tool support for MANET applications. We then *introduced the transmitter-receptor reflective model* that reconciles mirror-based reflection with ideas from classic communication-oriented reflective frameworks to the abstraction of an object references. This model provides a novel representation of remote object references in which *both* ends of a reference are reified by two dedicated metaobjects encapsulating all aspects of interactions between senders and receivers. We also revisit AmbientTalk/2's actor meta-level infrastructure to provide true structural correspondence for distribution, and to introduce an observer mechanism into actor mirrors which allows developers to be notified about the manipulation of an object by the interpreter without requiring an implicit mirror. We instantiated these meta-level mechanisms in *AmbientTalk/M*, a new dialect of AmbientTalk/2. The language is used in the rest of the dissertation as the software technology for enabling the development of failure handling abstractions and tool support for MANET applications.

- Chapter 6 introduced *ambient-oriented leasing*, which is our proposal for a object-oriented failure handling model suitable for MANETs. It is an object model that incorporates the concept of leasing at the heart of its design. We started this chapter by formulating four *criteria for a leasing model in a MANET*. We then made a proposal for expressing leasing as a programming language abstraction. We integrate our lease concept into remote object references giving rise to what we call *leased object references*: a time-decoupled object reference which deals with both transient and permanent failures. We then integrate the lease concept into future-type message passing giving rise to *due-type message passing*. We also discussed the effects of introducing those abstractions into a programming language and propose two novel abstractions to scope the effects of leased references and due-type messages: *leasing strategies* enable developers to assign the same *pass-by-leased-reference* semantics to a group of objects, and *leased message protocols* enable developers to assign the same timing assumptions to a group of messages.

- In the same chapter, we elaborated on the *open implementation of leased references*, and discuss how meta-level engineers can use the provided framework to express custom leased reference kinds, and build custom variations of leasing strategies and leased message protocols.

- In Chapter 7, we discussed a first integration of *ambient-oriented leasing into tuple spaces*. This resulted in a novel tuple space model called TOTAM. In the context of this dissertation, the key contributions of TOTAM lie with (1) extending a TOTA-like replication-based tuple space model with scoping mechanism to control the propagation of tuples in the network, and (2) integrating leasing into tuples enabling developers to determine upper boundaries on the availability of tuples in the system thereby avoiding the burden of manually removing the copies of a tuples (which is taken care by means of antituples). Moreover, the model also introduces a general programming concept in the form of a context rule to support development of context-aware applications in a mobile environment.

- Chapter 8 described how AmbientTalk/M, when extended with ambient-oriented leasing, satisfies our eight criteria for a failure handling model to be used in a MANET both in an object-oriented and tuple space-based distributed model.

- In Chapter 9, we *survey* the state of the art in distributed debugging techniques and tools. We concluded that current approaches are not suitable for debugging in a MANET setting because *they lack the necessary features to deal with the effects of partial failures* emerging from a radically different network characteristics than traditional, stationary networks.

- In Chapter 10, we identified two *challenges of debugging in the face of partial failures*. First, a debugger needs to be able to trace messages interchanged between communicating parties, leading to the concept of *message-oriented debugging*. Second, debugging sessions need to be open, and debuggers should be able to engage in running MANET applications, leading to the concept of *open debugging*. We henceforth refer to distributed debuggers that provide support for these challenges as *ambient-oriented debuggers*.

- In the same chapter, we presented the design and implementation of *an ambient-oriented debugger for AmbientTalk programs* called REME-D. REME-D's principal contribution is that it implements the features of ambient-oriented debuggers as an ambient-oriented application which incorporates a breakpoint-based debugging methodology where the focus is placed on the exchange of asynchronous messages between actors. REME-D also shows that it is feasible to implement an ambient-oriented debugger reflectively using a mirror-based reflective architecture with the enhancements of Chapter 5.

- In Chapter 11, we presented the design and results of a *pre-experimental user study* conducted for validating REME-D. Although the results of such an experiment are not conclusive, the study provided us with three valuable insights with regard to how real users perceive and value the features of an ambient-oriented debugger: (1) the features that users actually expect from an ambient-oriented debugger were indeed supported in REME-D, (2) users value REME-D as a tool to make ambient-oriented programming easier, and (3) further work is required on the UI as it had a great impact on how users perceive and value the features of REME-D.

## 12.3 Limitations

In the preceding chapters we already highlighted specific technical limitations of the approaches they describe. In this section, we recall the most important ones, and position them in the broader context of this dissertation.

### 12.3.1 Meta-level Engineering in AmbientTalk

Our transmitter-receptor model aims to provide a good reflective model where to represent custom referencing abstractions. As a result, our solution has been shaped towards expressing distributed interactions based on an object-oriented programming approach, in which objects are passed by object reference among communicating parties. It remains a future question to extend such a model to AmbientTalk's *isolates*, objects that adhere to pass-by-copy semantics. For example, the implementation of lazy references (cf. Section 5.2.4) would benefit from further control over object serialization protocol.

We should also investigate the performance impact of the transmitter-receptor model and the other meta-level abstractions introduced in AmbientTalk/M. Although the current implementation already limited the reification of some meta-level operations to object references with custom receptors, further work is required to optimize our language. This is relevant because our current experimental setup for AmbientTalk applications is Android devices. While these devices are getting more powerful in the recent years, they still remain constrained in comparison to full-fledged computers.

### 12.3.2 Ambient-Oriented Leasing

This dissertation has proposed programming language support that offers developers fine-grained scoping abstractions for leasing. However, so far, we have mainly focused on *dynamic* scoping constructs for leased references and leased messages. This means that, for example, acquiring a leased reference to a vector, does not have any implication for the individual leases to the objects within the vector. Exploring data scoping for leasing to provide "leased-based data structures" remains an open research question. While we do not see fundamental issues to express such abstractions based on our transmitter-receptor model, it may require revisiting the reflective API exposed by leased references.

Another design choice when exploring scoping for leasing constructs is that we opted for *explicit* scoping abstractions. While leasing strategies and leased message protocols solve the problem of *scattering* leasing in the application code, leasing remains *entangled* with application code. An alternative to these scoping abstractions would be to introduce techniques for separation of concerns such as those provided by aspect-oriented programming. However, leasing requires an expressive pointcut language (which can capture parameter passing of objects and messages) in which leasing concerns may evolve independently from the base functionality, exacerbating the pointcut fragility problem. Endorsing the arguments against distribution transparency stated in the introduction, we believe that explicit scoping abstractions outweigh the benefits of full modularization.

Finally, we would like to note that composition of leased abstractions has not been explored in the current leasing model. An important aspect of the current implementation is that the different interaction policies between leased references and leased messages described in Section 6.6 are encoded in the underlying transmitter-receptor pair. As such, it remains to extend our open implementation with means to allow developers to encode custom interaction policies. Moreover, we also encode a default strategy for nesting leasing strategies and leased message protocols which cannot be customized by ambient-oriented leasing meta-level engineers.

### 12.3.3 Ambient-Oriented Leasing for Tuple Spaces

The most significant limitation of TOTAM lies with its leasing model. As previously mentioned, TOTAM remains a first integration of ambient-oriented leasing and as such, the abstractions incorporated for leasing need to be further developed and subjected to more experimentation. In particular, we need to study how we can translate useful leasing patterns present in our object-oriented leases (such as renew-on-call leased references), to the realm of tuples.

Experiences with students using TOTAM (in the context of our university's Distributed and Mobile Programming Paradigms course) have shown the importance of linguistic support for monitoring the network connectivity of the devices forming the underlying TOTAM network. While this may defeat the metaphor of a shared tuple space, it appears to be a necessary feature which is currently encoded by hand in TOTAM (as we discussed in Section 8.1). Recall from Chapter 7 that programmers need to encode this in TOTAM by injecting leased tuples which "simulate" the presence of a device in the network. It remains to be seen how to elegantly integrate network-awareness support in the TOTAM model. We are considering a solution like in LIME which adds a "meta" tuples space providing such information.

### 12.3.4 Ambient-Oriented Debuggers

In order to complete the study of partial failures in the context of tool support, we need to integrate our debugger with the rest of language constructs for ambient-oriented leasing. In short, the current incarnation of REME-D can only interact with future-type messages. However, the debugger cannot control the leasing semantics of leased references and due-type messages. As a result, developers cannot "freeze" leases, expire or renew leased references at will. It remains to be investigated what it means to debug a lease, and which are the appropriate features that a debugger needs to provide to this end. In a broader context, the exploration of debugging reflective language constructs is an open issue left for future research.

Since the debugger is itself an ambient-oriented application, its *implementation* could also benefit from leased references. Currently, we assume a single debugging session, and this session is not leased. As such, if an actor disconnects from the debugging session, the debugging facilities are deactivated, and the corresponding data structures need to be cleaned up by hand. Employing leasing in the debugger implementation is essential in order to provide advanced debugging features such as multiple debugging sessions or back-in-time debugging in an efficient way.

## 12.4 Avenues for Future Research

In this section, we discuss how our research could be extended or studied in a different context, rather than focusing on the technical limitations of its current instantiation.

### 12.4.1 Towards a Distributed Secure Object Model

Security has been been left unexplored in the context of ambient-oriented programming so far. With the emergence of new kinds of mobile applications dealing with "digital money", proposing a secure distributed model is more crucial than ever. These mobile applications running on the user's smartphone represent the client's *virtual wallet* storing digital money. Proposals such as BitCoin [Nak09] exist that allow money to be safely transferred in a peer-to-peer distributed network. However, this only involves numerical information. The digital currencies envisioned for the future "virtual wallet-based mobile applications" may represent a wide spectrum of information including digital coupons, rides on a metro card, concert tickets, points in loyalty cards, etc. Such rich types of information actually need a full-fledged object-oriented representation. For example, a coupon has graphical information, expiry date, usage restrictions (e.g., not to be used with other coupons) and so on.

How to conceive and provide a secure distributed object-oriented model for easing the construction of such mobile applications remains an open research question. We believe that leased references provides a solid basis to build a distributed object-oriented model that integrates security mechanisms (e.g., avoid unauthorized access to objects, forging of objects, and duplication of objects, etc.) without requiring fixed infrastructure. We are currently thinking of combining them with language-based security techniques such as capability-based security [Mil06]. As discussed in Section 5.5, the object-capability model is a suitable model for securing distributed interactions in AmbientTalk/M because it uses the object reference graph as the access control graph. Recall from Section 6.2.2 that our notion of a lease already includes a component that allows us to control which kind of access third parties can gain to a leased service. This sets the basis for abstractions that limit the spread of leased references and enforce *the principle of least authority* promoted by the object-capability model.

### 12.4.2 Structuring the Object Soup

A main contribution of this dissertation is the study of a remote object reference abstraction that incorporates machinery for dealing with both transient and permanent failures. However, as objects are parameter passed back and forth during distributed interactions, the object graph becomes increasingly complex and distributed across different devices, forming an unstructured "soup" of objects. We believe that a relevant open question in the context of ambient-oriented programming is now to build abstractions to structure such an object soup.

We have initial ideas on how to impose structure on those graphs by combining ideas from ownership types with the transmitter-receptor model embodied in AmbientTalk/M. Ownership types [CPN98] were initially devised as a static typing mechanism to avoid problems caused by sharing objects through aliasing. They impose structure on object graphs by putting objects into boxes (represented by the *owner*), limiting the visibility of object references and restricting access from other boxes using *context parameters*. The owner of the object is not necessarily the object that creates it, thus decoupling the concept of owning an object and having a reference to it. Using AmbientTalk/M, one could already integrate custom referencing abstractions that restrict the way how objects in different devices can access each other. A receptor can be seen as the *box* that encapsulates all references handed out to other devices, and the entity used to limit how the references can be used (by checking the validity of each message sent through it). However, it remains to be seen which context parameters are relevant in an ambient-oriented context and whether our API is robust enough to express them. In Listing 5.2, we showed the prototype implementation of a safe reference exhibiting semantics akin to an immutable reference in [CWÖJ08] and "arg" reference in [NVP98] that prevents a client object from calling methods that mutate a target object. Such a reference was built by overriding the `performInvocation` method to prevent synchronous method invocations executed by the target object. However, to date, no ownership type system has been applied to a distributed software platform.

### 12.4.3 Debugging Distributed Asynchronous Applications

Finally, it remains to be studied whether the principles of an ambient-oriented debugger can be translated to other languages and software platforms. In particular, an interesting avenue of research is to investigate how the features of an ambient-oriented debugger can help us to understand Ajax-enabled web applications and their debugging process.

Ajax [Gar05] programs work as communicating event loops processing user interface events as well as asynchronous messages from a server, or from other JavaScript event loops within the browser. Unfortunately, debugging tools currently available such as Firebug, or Chrome's debugger only support debugging of sequential client's code (browser) providing very little control over the requests sent to the server. It is precisely this inter-process communication that is essential to understand the behaviour of a distributed application. Similar to MANET applications, Ajax-enabled web applications do require message-oriented debugging.

Moreover, there exist some JavaScript libraries such as Q that provide support for future-type message passing. To the best of our knowledge, no debugging support is provided for those libraries. Stepping commands such as the step-return or step-into described in Section 10.3.3, thus become really handy when debugging future-type message passing interactions. In conclusion, exploring and translating the set of principles present in an ambient-oriented debugger and REME-D to the realm of web-based applications poses an interesting research question.

## 12.5 Concluding Remarks

This work can be seen as the continuation of the activities of ambient-oriented programming research group, the goal of which is to build "nec plus ultra" distributed programming technology for the development of MANET applications. In the introductory chapter, we discussed the gap between the AmOP *optimistic model* which makes network disconnections completely invisible to the programmer, and the traditional distributed *pessimistic model* in which network disconnections result in exception handling. Our research closes that gap by augmenting an optimistic model with the suitable failure handling abstractions that allow developers to detect, reason about and handle partial failures at the programming level. We claim that this results in a *realistic model* to deal with the effects of partial failures. While the abstractions and tools described in this thesis are research experiments, we argue that they form the basis of a solid software development platform for supporting the development of MANET applications.

# Appendix A

# Code Listing of the Mobile Music Player

This appendix includes the core functionality of the mobile music player application as discussed in Section 6.8. We first include the implementation of the application in Java RMI and then in AmbientTalk. Table A.1 summarizes the colors used in the code corresponding to the four concerns used to evaluate the application (described in Section 6.8.2), and the service discovery code.

| Color | Code deals with... |
|---|---|
| blue | memory management |
| red | concurrency control |
| green | failure handling |
| purpule | application |
| none | service discovery (not included in the quantitative evaluation) |

Table A.1: Code Legend.

## A.1 Code Listing of the Java RMI Implementation

This section includes the Java RMI implementation of the music player application according to the UML class diagram depicted in Figure 6.8. Each of the following sections contains the source code listing for each class.

### A.1.1 MusicPlayer

```java
package tools.musicPlayer;

import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Timer;
import java.util.TimerTask;
import java.util.Vector;

public class MusicPlayer {
  public static final String RMI_REGISTRY = "192.168.1.100";
  public static final String MUSIC_PLAYER = "MusicPlayer";
  public static final int LEASE_SESSION_INTERVAL = 60000; // 1 minute
  public static final int THRESHOLD = 25;
  public String username_;
  public Vector myLib_;
  // transmissionActor in charge of the remote message sending of this session.
  private HashMap sessions_;
  public MusicPlayer(String username){
    username_ = username;
    myLib_ = new Vector();
    sessions_ = new HashMap();
  }

  public class RemoteInterface implements iMusicPlayer {

    public RemoteInterface() {}

    public iLeaseSession openSession(final String remoteUser)
    throws RemoteException{
      System.out.println("opening new session for " + remoteUser);
      Vector senderLib = new Vector();
      Session session = new Session(remoteUser, this);
      LeaseSession leaseSession=new LeaseSession(session,LEASE_SESSION_INTERVAL);
      leaseSession.whenExpired(new ExpirationListener() {
        public void leaseExpired() {
          System.out.println("session with " + remoteUser + " timed out.");
        }
      });
      return leaseSession;
    }
    public int getSizeOfLibrary() throws RemoteException{
      return myLib_.size();
    }
    //added to access library from session called always local.
    public Vector getLib(){
      return myLib_;
    }
  }

  public synchronized void sendSong(
    iLeaseSession session, int index, CallbackActor callbackActor)
  {
```

```
  Actor transmissionActor = (Actor) sessions_.get(session);
  if (index < 0) {
    // callback from endExchange or an timeoutException on uploadSongMsg occured
    // stop the session and make sure that an unreferenced message is sent by:
    // -stopping transmissionActor (in the AckReturnValueMsg)
    // -removing session from map (must be done here)
    transmissionActor.stopProcessing();
    sessions_.remove(session);
  }else{
    if (myLib_.size() > index) {
      Song song = (Song) myLib_.get(index);
      //annotate the message with @Due(leaseTimeLeft: session)
      //Note: to be able to do this we implement the client side of the lease,
      //otherwise, it will require another remote method call.
      long timeout = ((ClientLeaseSession)session).getTimeLeft();
      UploadSongMsg songMsg = new UploadSongMsg(
        song, index +1, timeout, callbackActor);
      songMsg.whenExpired(new ExpirationListener(){
        public void leaseExpired() {
          System.out.println("stopping session ");
        }
      });
      transmissionActor.receive(songMsg);
    }else{
      transmissionActor.receive( new EndExchangeMsg());
    }
  }
};
public synchronized void receiveSession(iLeaseSession session,
   TransmissionActor transmissionActor, CallbackActor callbackActor
){
  //associate the session with a transmission actor
  if (session != null){
    sessions_.put(session, transmissionActor);

    sendSong(session, 0, callbackActor);
  }
}
public synchronized void whenDiscovered(iMusicPlayer remotePeer)
  throws InterruptedException
{
    // discover other music players
  CallbackActor callbackActor = new CallbackActor(this);
  TransmissionActor transmissionActor = new TransmissionActor(callbackActor);
  //annotate the message with @Due(1 minute)
  OpenSessionMsg msg = new OpenSessionMsg(
    remotePeer, username_, 60000, callbackActor);
  transmissionActor.receive(msg);
  msg.whenExpired(new ExpirationListener(){
    public void leaseExpired() {
      System.out.println("unable to open a session");
    }
  });
  }
public void goOnline(String remoteUser){
    //export remoteInterface as MusicPlayer + username
  //(which is used in rmi as unique identifier for service)
  try {
      iMusicPlayer stub = (iMusicPlayer) UnicastRemoteObject.exportObject(
      new RemoteInterface(), 0);
      // Bind the remote object's stub in the registry
      Registry registry = LocateRegistry.getRegistry(RMI_REGISTRY);
      registry.bind(MUSIC_PLAYER + username_, stub);
      System.out.println("Remote interface exported");
    } catch (Exception e) {
      System.err.println("Export:as exception: " + e.toString());
      e.printStackTrace();
    }
```

```
      // start discover other music players
    // Using Timer and DiscoveryTimerTask to detect other music players
      // rmi throws an exception if the service is not yet in the rmi.
      Timer discoveryDetectorTimer = new Timer();
      DiscoveryTimerTask discoveryDetector =
      new DiscoveryTimerTask(discoveryDetectorTimer, remoteUser);
      discoveryDetectorTimer.scheduleAtFixedRate(
      discoveryDetector, 0, DiscoveryTimerTask.DISCOVERY_RATE);
  }
```

```
  public void addSong(String artist, String title){
    myLib_.add(new Song(artist, title));
  }
```

```
  private class DiscoveryTimerTask extends TimerTask{
    /**
     * The rate at which to schedule this timer task
     */
    public static final int DISCOVERY_RATE = 3000; // in milliseconds
    public String service_;
    public boolean found_;
    public Timer discoveryDetectorTimer_;

    public DiscoveryTimerTask(Timer timer, String service){
      discoveryDetectorTimer_ = timer;
      service_ = service;
      found_ = false;
    }

    public void run(){
      try {
        if (found_ == false){
          Registry registry = LocateRegistry.getRegistry(MusicPlayer.RMI_REGISTRY);
          iMusicPlayer peer_ = (iMusicPlayer) registry.lookup(
            MusicPlayer.MUSIC_PLAYER + service_);
          discoveryDetectorTimer_.cancel();
          try {
              System.out.println("discovered a new music player: " + service_);
            whenDiscovered(peer_);
          } catch (InterruptedException e) {
            System.out.println("Unknown whenDiscovered exception " + e.toString());
            e.printStackTrace();
          }
        }else {System.out.println("found! I shouldn't be here!");}
      } catch (NotBoundException e){
        // the service is not found yet.
      } catch (Exception e) {
        System.err.println("Unknown discoveryTimer exception: " + e.toString());
        e.printStackTrace();
      }

    }
  }
}
```

### A.1.2 iMusicPlayer

```
package tools.musicPlayer;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface iMusicPlayer extends Remote {
    public iLeaseSession openSession(final String remoteUser) throws RemoteException;
    public int getSizeOfLibrary() throws RemoteException;
}
```

### A.1.3 Session

```
package tools.musicPlayer;

import java.rmi.RemoteException;
import java.util.Vector;

import tools.musicPlayer.MusicPlayer.RemoteInterface;

public class Session{

  public String remoteUser_;
  public Vector senderLib_;
  public RemoteInterface mp_;

  protected Session(String remoteUsername, RemoteInterface mp) {
    senderLib_ = new Vector();
    remoteUser_ = remoteUsername;
    mp_ = mp;
  }

  public String uploadSong(String artist, String title) throws RemoteException {
    senderLib_.add(new Song(artist, title));
    return "ok";
  }

  public void endExchange() throws RemoteException{
    Vector myLib = mp_.getLib();
    senderLib_.retainAll(myLib);
    int matchRatio = (int) ((senderLib_.size()*100)/(myLib.size()+0.01));
    if (matchRatio >= MusicPlayer.THRESHOLD) {
      System.out.println("Found user "+ remoteUser_ +
      " with similar taste in music ("+matchRatio+"% match)");
      //notifyOnMatch
    } else {
      System.out.println("User "+ remoteUser_+ "
      does not share your taste in music ("+matchRatio+"% match)");
    };
    //"done";
  }
}
```

### A.1.4 Song

```
package tools.musicPlayer;

public class Song {

  public String artist_;
  public String title_;
  public int timesPlayed_;

  public Song(String artist, String title){
    artist_ = artist;
    title_ = title;
    timesPlayed_ = 0;
  }
  public void play(){
    timesPlayed_ ++;
  }
  public boolean equals(Object other){
    return (other != null) && (this.getClass() == other.getClass()) &&
    (artist_.equals(((Song) other).artist_) &&
    (title_.equals(((Song) other).title_)));
  }
  public String toString(){
    return artist_ + " - " + title_ + "(" + timesPlayed_ + ")";
  }
}
```

### A.1.5 ExpirationListener

```
package tools.musicPlayer;

public abstract class ExpirationListener {
  public abstract void leaseExpired();
}
```

### A.1.6 Message

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;

public abstract class Message {
  // Messages can be annotated with @Due(timeout).
  // Following fields are required for that interaction
  public long timeout_;
  public Lease dueLease_;
  public boolean expired_;
  public ExpirationListener listener_;

  public Message(long timeInterval){
    expired_ = false;
    if (timeInterval > 0 ){
      dueLease_ = new Lease(timeInterval);
    }
  }
  public void whenExpired(ExpirationListener listener){
    // one expired per message send - for this application is fine.
    if (listener_ == null) {
      listener_ = listener;
    }
  }
  public boolean isExpired(){
    return expired_;
  }
  public void expire(){
    expired_ = true;
  }
  public abstract void process(Actor a) throws NoSuchObjectException;
}
```

### A.1.7 OpenSessionMsg

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;

public class OpenSessionMsg extends Message {
  public iMusicPlayer peer_;
  public String username_;
  CallbackActor callbackActor_;
  public  OpenSessionMsg(iMusicPlayer remotePeer, String username,
      long timeoutDue, CallbackActor callbackActor){
    super(timeoutDue);
    if (timeoutDue>0) {
      dueLease_.whenExpired( new ExpirationListener(){
        public void leaseExpired() {
          expired_ = true;
          if (listener_ != null) listener_.leaseExpired();
          //sending an empty expired message to be able to stop callbackActor.
          SessionReturnValueMsg expiredMsg = new SessionReturnValueMsg(null, null);
          expiredMsg.expire();
          callbackActor_.receive(expiredMsg);
        }
      });
    }
    username_ = username;
    peer_ = remotePeer;
    callbackActor_ = callbackActor;
  }
  public void process(Actor a) throws NoSuchObjectException {
    try {
      iLeaseSession session = peer_.openSession(username_);
      if (!expired_){
        dueLease_.revoke();
        ClientLeaseSession leaseSession =
          new ClientLeaseSession(session, MusicPlayer.LEASE_SESSION_INTERVAL);
        a.setSession(leaseSession);
        callbackActor_.receive(new SessionReturnValueMsg(
              leaseSession, (TransmissionActor) a));
      }
    } catch (NoSuchObjectException e0){
      throw e0;
    } catch (RemoteException e) {
      //exception while session active, reschedule to simulate buffering.
      if (!expired_){
        a.receivePrioritized(new OpenSessionMsg(
          peer_,username_,timeout_,callbackActor_));
        e.printStackTrace();
      }
    }
  }
}
```

## A.1.8 UploadSongSessionMsg

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;

public class UploadSongMsg extends Message{
  public Song song_;
  public int index_;
  CallbackActor callbackActor_;

  public UploadSongMsg(Song song, int index, long timeoutDue,
      CallbackActor callbackActor){
    super(timeoutDue);
    if (timeoutDue>0) {
      dueLease_.whenExpired( new ExpirationListener(){
        public void leaseExpired() {
          expired_ = true;
          if (listener_ != null) listener_.leaseExpired();
          // by sending -1 we make sure that the callbackActor stops and
          // the used infrastructure for the session in MusicPlayer is cleaned.
          callbackActor_.receive(new AckReturnValueMsg(-1));
          System.out.println("TIMEOUT song " + song_.toString());
        }
      });
    }
    song_ = song;
      index_ = index;
      callbackActor_ = callbackActor;
  }
  public void process(Actor a) throws NoSuchObjectException {
    CallbackActor callbackActor=(CallbackActor)((TransmissionActor) a).getOwner();
    iLeaseSession session = a.getSession();
    try {
      String ack = session.uploadSong(song_.artist_, song_.title_);
      //check again for expired_ because the TransmissionActor thread may be
      //blocked when the message already expired.
      if (!expired_){
        dueLease_.revoke();
        callbackActor.receive(new AckReturnValueMsg(index_));
        System.out.println("acked sent song " + song_.toString());
      };
    } catch (NoSuchObjectException e0){
      // trying to use an expired session, throw again.
      throw e0;
    } catch (RemoteException e) {
      //exception while session active, reschedule to simulate buffering.
      if (!expired_){
        System.out.println("reschedule message-unable to send" +song_.toString());
        a.receivePrioritized(new UploadSongMsg(
            song_, index_, timeout_, callbackActor_));
        e.printStackTrace();
      }
    }
  }
}
```

## A.1.9 EndExchangeMsg

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;

public class EndExchangeMsg extends Message {

  public EndExchangeMsg() {
    super(0); // Msg without timeout.
  }

  public void process(Actor a) throws NoSuchObjectException {
    CallbackActor callbackActor=(CallbackActor)((TransmissionActor) a).getOwner();
    iLeaseSession session = a.getSession();
    try {
      session.endExchange();
      //callback from endExchange is required in Java because
      // the client has to let rmi know that the reference is not used anymore,
      // i.e. make sure that the unrefenced control message is sent.
      // clean the used infrastructure for the session by sending -1.
      callbackActor.receive(new AckReturnValueMsg(-1));
      a.stopProcessing();
    } catch (NoSuchObjectException e0){
      throw e0;
    } catch (RemoteException e) {
      //exception while session active, reschedule to simulate buffering.
      System.out.println("reschedule message - unable to end exchange " );
      a.receivePrioritized(new EndExchangeMsg());
      e.printStackTrace();
    }
  }
}
```

### A.1.10 AckReturnValueMsg

```java
package tools.musicPlayer;

public class AckReturnValueMsg extends Message {
  public int index_;
  public AckReturnValueMsg (int index){
    super(0);
    index_ = index;
  }
  public void process(Actor a) {
    iLeaseSession session = a.getSession();
    MusicPlayer callback = (MusicPlayer) ((CallbackActor) a).getOwner();
    callback.sendSong(session, index_, (CallbackActor) a);
    if (index_ < 0 ) a.stopProcessing();
  }
}
```

### A.1.11 SessionReturnValueMsg

```java
package tools.musicPlayer;

public class SessionReturnValueMsg extends Message{

  public iLeaseSession session_;
  public TransmissionActor ta_;
  public SessionReturnValueMsg (iLeaseSession session, TransmissionActor ta){
    super(0);
    session_ = session;
    ta_ = ta;
  }
  public void process(Actor a) {
    a.setSession(session_);
    MusicPlayer callback = (MusicPlayer) ((CallbackActor) a).getOwner();
    callback.receiveSession(session_, ta_, (CallbackActor) a);
  }
}
```

### A.1.12 EventLoop

```java
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.util.Vector;

class MessageQueue{
    private static final int _DEFAULT_QUEUE_SIZE_ = 10;

  private final Vector elements_;

  public MessageQueue() {
    elements_ = new Vector(_DEFAULT_QUEUE_SIZE_);
  }
  /**
   * Enqueue an event in the buffer. This method wakes up any
   * waiting consumer threads.
   */
  public synchronized void enqueue(Message msg) {
    elements_.add(msg);
    notify();
  }

  public synchronized void enqueueFirst(Message msg) {
    elements_.add(0, msg);
    notify();
  }
  /**
   * Dequeue a message from the queue.
   * This method will block when the buffer is empty!
   */
  public synchronized Message dequeue() {
    try {
      while(elements_.isEmpty()) {
        wait();
      }
      return (Message) elements_.remove(0);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return null;
  }
}
public abstract class Actor implements Runnable{

  protected MessageQueue messageQueue_;
    public iLeaseSession session_;
    protected boolean askedToStop_;
    protected Thread thread_;

  public Actor(){
    messageQueue_ = new MessageQueue();
    askedToStop_ = false;
    session_ = null;
    thread_ = new Thread (this);
    thread_.start();
  }
  public void receive(Message msg) {
    messageQueue_.enqueue(msg);
  }
  /*
   * When an actor receives a message, it is immediately placed
   * in the message queue and will be processed later.
   * Using this method, messages are scheduled first in the queue.
   * Used by to retransmit remote message call that failed, so that
   * we provide similar buffering of messages than in AT.
   */
  public void receivePrioritized(Message msg) {
    messageQueue_.enqueueFirst(msg);
  }

  public void stopProcessing() {
    if (!askedToStop_) {
      askedToStop_ = true;
      // explicitly interrupt my event processor because it
      // may be blocked waiting on other events
      thread_.interrupt();
      // to provoke the unreference control message.
      session_ = null;
    }
  }

  public void handleMessage(Message msg) throws NoSuchObjectException{
    msg.process(this);
  };
  public void run(){
    while(!askedToStop_){
      try {
        Thread.sleep(1000);
        Message msg = messageQueue_.dequeue();
        if (!msg.isExpired()) {
          handleMessage(msg);
        } else{
          //make sure that this threads stops
          this.stopProcessing();
        }
      } catch (Exception e) {
        System.out.println("error while handling a message");
        e.printStackTrace();
      }
    }
    System.out.println("got interrupted" + this.toString());
  }

  public abstract Object getOwner();

  public iLeaseSession getSession(){
    return session_;
  }
  public void setSession(iLeaseSession session){
    session_ = session;
  }
}
```

### A.1.13 ELCallback

```
package tools.musicPlayer;

public class CallbackActor extends Actor {

  public MusicPlayer owner_;

  public CallbackActor(MusicPlayer mp){
    super();
    owner_ = mp;
  }
  public Object getOwner(){
    return owner_;
  }
  public String toString(){
    return "CallbackActor";
  }
}
```

### A.1.14 ELTransmission

```
package tools.musicPlayer;

public class TransmissionActor extends Actor {

  public CallbackActor owner_;

  public TransmissionActor(CallbackActor actor){
    super();
    owner_ = actor;
  }
  public Object getOwner(){
    return owner_;
  }
  public String toString(){
    return "TransmissionActor";
  }
}
```

### A.1.15 iLeaseSession

```
package tools.musicPlayer;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface iLeaseSession extends Remote{
  public String uploadSong(String artist, String title) throws RemoteException;
  public void endExchange()throws RemoteException;
}
```

### A.1.16 LeaseSession

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.server.Unreferenced;

public class LeaseSession extends UnicastRemoteObject
    implements iLeaseSession, Unreferenced{

  public Session session_;
  public Lease lease_;

  public LeaseSession(Session session, long leaseInterval) throws RemoteException{
    lease_ = new Lease(leaseInterval);
    session_ = session;
  }
  public String uploadSong(String artist, String title) throws RemoteException {
    if (lease_.isExpired()) {
      throw new NoSuchObjectException("asked for an expired session");
    } else {
      lease_.renew();
      return session_.uploadSong(artist, title);
    }
  }
  public void endExchange() throws RemoteException {
    if (lease_.isExpired()) {
      throw new NoSuchObjectException("asked for an expired session");
    } else {
      lease_.revoke();
      session_.endExchange();
    }
  }
  public void whenExpired(ExpirationListener listener){
    lease_.whenExpired(listener);
  }
}
```

### A.1.17 ClientLeaseSession

```
package tools.musicPlayer;

import java.rmi.NoSuchObjectException;
import java.rmi.RemoteException;

public class ClientLeaseSession extends Lease implements iLeaseSession {

    public iLeaseSession session_;

  public ClientLeaseSession(iLeaseSession session, long leaseInterval) {
    super(leaseInterval);
    session_ = session;
  }
  public String uploadSong(String artist, String title) throws RemoteException {
    if (isExpired()) {
      throw new NoSuchObjectException("asked for an expired session");
    } else {
      this.renew();
      return session_.uploadSong(artist, title);
    }
  }
  public void endExchange() throws RemoteException {
    if (isExpired()) {
      throw new NoSuchObjectException("asked for an expired session");
    } else {
      revoke();
      session_.endExchange();
    }
  }
  public void renew(){
    System.out.println("renewing client lease");
    super.renew();
  }
}
```

## A.1.18 Lease

```
package tools.musicPlayer;

import java.util.Iterator;
import java.util.Timer;
import java.util.TimerTask;
import java.util.Vector;

public class Lease {

  public long leaseInterval_;
  private Timer leaseTimer_;
  private LeaseTimerTask timerSubscription_;
  protected boolean expired_;
  private Vector whenExpiredListeners_; //lazy initialization

  public Lease(long leaseInterval){
    leaseInterval_ = leaseInterval;
    expired_ = false;
    leaseTimer_ = new Timer();
    timerSubscription_ = new LeaseTimerTask();
    leaseTimer_.schedule(timerSubscription_, leaseInterval);
  }
  public synchronized void expire(){
  //  System.out.println("expiring lease");
    revoke();
    //notify expiration to listeners.
    if (whenExpiredListeners_ != null) {
      for(Iterator iterator=whenExpiredListeners_.iterator();iterator.hasNext();){
        ((ExpirationListener)iterator.next()).leaseExpired();
      }
    }
  }
  public synchronized void revoke(){
    //System.out.println("revoking lease");
    if (!isExpired()){
      expired_ = true;
      leaseTimer_.cancel();
    }
  }
  public synchronized boolean isExpired(){
    return expired_;
  }
  public void renew(){
    if (!isExpired()){
      timerSubscription_.cancel();
      timerSubscription_ = new LeaseTimerTask();
      leaseTimer_.schedule(timerSubscription_, leaseInterval_);
    }
  }
  public void whenExpired(ExpirationListener listener){
    if (isExpired()){
      listener.leaseExpired();
    } else{
      if (whenExpiredListeners_ == null) {
        whenExpiredListeners_ = new Vector(1);
      }
      whenExpiredListeners_.add(listener);
    };
  }
  public long getTimeLeft(){
    return timerSubscription_.getTimeRemaining();
  }
  private class LeaseTimerTask extends TimerTask{
    public boolean cancelled_;
    public Timer timer_;
    public LeaseTimerTask(){
      cancelled_ = false;
    }
    public void run() {
      if (!cancelled_) expire();
    }
    public long getTimeRemaining(){
      return this.scheduledExecutionTime() - System.currentTimeMillis();
    }
    public boolean cancel(){
      cancelled_ = true;
      return super.cancel();
    }
  }
}
```

## A.2 Code Listing of the AmbientTalk Implementation

```
def AmbientRefsM := /.at.lang.ambientrefs_old; // for discovery
deftype MusicPlayer; // music players are exported using this service type
import /.at.lang.futures; // for when:becomes:
enableFutures(false);
import /.at.lang.leasedrefs exclude minutes, seconds, millisec; // for leasing
def Song := object: {
  def artist := nil;
  def title := nil;
  def timesPlayed := 0;
  def init(artist, title) {
    self.artist := artist;
    self.title := title;
    self.timesPlayed := 0;
  };
  def ==(other) {
    (artist == other.artist).and: {title == other.title};
  };
  def play() {
    timesPlayed := timesPlayed + 1;
  };
  def toString() {
    artist + " - " + title + "(" + timesPlayed + ")";
  };
};
def Vector := jlobby.java.util.Vector; // we represent song libraries as vectors
def THRESHOLD := 25; // when users share 25% of their songs, we signal a match
def MusicPlayerModule := object: {
  def myLib := Vector.new(); // the local user's songs library
    def userName := jlobby.java.lang.System.getProperty("user.name");
    def notifyOnMatch := { |user, percentage| nil };
    def init(userName, notifier := notifyOnMatch) {
    self.userName := userName;
      myLib := Vector.new();
      notifyOnMatch := notifier;
  };
  def notification(@texts) {
    system.println("[music player "+userName+"] ", @texts);
  };
  def createInterface() {
    object: {
      def openSession(remoteUser) {
        notification("opening new session for " + remoteUser);
        def senderLib := Vector.new();
        def session := object: {
          def uploadSong(artist, title) {
            senderLib.add(Song.new(artist, title));
            "ok"; // tell sender that song was successfully received
          };
          def endExchange() {
            revoke: session; // takes the session offline
            senderLib.retainAll(myLib);
            def matchRatio := (senderLib.size()*100/(myLib.size()+0.01)).round();
            if: (matchRatio >= THRESHOLD) then: {
              notification("Found user ", remoteUser,
              " with similar taste in music (",matchRatio,"% match)");
              notifyOnMatch(remoteUser, matchRatio);
            } else: {
              notification("User ", remoteUser, "
              does not share your taste in music (",matchRatio,"% match)");
            };
          };
        };
          def leasedSession := renewOnCallLease: 10.minutes for: session;
          when: session expired: {
            notification("session with " + remoteUser + " timed out.");
            senderLib := nil;
          };
```

```
     // return session object (which will be leased referenced) to client
     leasedSession;
    }
    def getSizeOfLibrary() { myLib.size() };
   };
  };
 def goOnline() {
  export: createInterface() as: MusicPlayer;
   // uses an ambient reference to discover one other music player
   def musicPlayerFuture := AmbientRefsM.ambient: MusicPlayer;
   when: musicPlayerFuture becomes: { |ambientReference|
   notification("discovered new music player: " + ambientReference);
   def futureSession := ambientReference<-openSession(userName)@Due(1.minutes);
    when: futureSession becomes: {  |session|
    def iterator := myLib.iterator(); // to iterate over own music library
    def sendSongs() { // auxiliary function to send each song
     if: (iterator.hasNext()) then: {
      def song := iterator.next();
      def futureUploadSong := session<-uploadSong(song.artist,
       song.title)@Due(leaseTimeLeft: session);
      when:  futureUploadSong becomes: { |ack|
       notification("sent song " + song.artist + " - " + song.title);
       sendSongs(); // recursive call to send the rest of the songs
      } catch: { |exception|
       notification("stopping exchange: " + exception)
      };
     } else: {
      session<-endExchange()@OneWayMessage;
     };
     nil;
    };
    sendSongs();
   } catch: TimeoutException using: { |e|
    notification("unable to open a session");
   }

  };
 };
 def addSong(artist, title) {
  myLib.add(Song.new(artist, title));
 };
};
```

# Appendix B

# REME-D's User Study Material

This appendix includes the material created for the pre-experimental user study for REME-D. In particular, it provides the questionnaires designed for the pretest and posttest, and the debugging assignment given to the participants during the experiment.

## B.1 Pretest Questionnaire

In the experiment conducted to evaluate REME-D , two questionnaires were used. This section prints the pretest questionnaire used to get a zero-measurement before the participants started their assignments with the tool. We first show the questions according to the different topics the participants were inquired. Afterwards, we include the questions in the original form used in the experiment. A more detailed explanation of each question's purpose is given in Section 11.2.1.

| | |
|---|---|
| **Development Experience** | |
| A | I consider myself an experienced developer. |
| B | I am proficient in development distributed systems. |
| C | I understand the principles of ambient oriented programming in AmbientTalk. |
| E | I am familiar with the Eclipse IDE. |
| K | I am proficient in the use of online (breakpoint-based) debuggers. |
| **Attitude towards AmbientTalk** | |
| D | Developing in AmbientTalk is easy. |
| **Attitude towards Eclipse IDE** | |
| F | Eclipse is a good IDE to develop AmbientTalk programs. |
| **Attitude towards debugging** | |
| G | A lot of bugs can be prevented by using better development tools. |
| H | Debugging distributed programs is hard. |
| I | Debuggers are a helpful tool to find errors in programs. |
| J | Debuggers are a helpful tool to understand programs. |
| P | A debugger for AmbientTalk is needed. |
| **Expectations of a debugger for AmbientTalk programs** | |
| O | A debugger would make programming AmbientTalk easier. |
| L | Breakpoints are essential to debugging a program. |
| M | Step-by-step execution is essential to debugging a program. |
| N | Inspecting the program’s state is essential to debugging a program. |
| Q | When debugging AmbientTalk programs messages between actors are more important than messages between objects. |
| R | Inspecting mailbox contents is essential to debugging AmbientTalk programs. |
| S | An AmbientTalk debugger must take into account the frequent disconnections and zero infrastructure phenomenon present in a mobile ad hoc networking setting. |

Table B.1: Pretests questions according to the five topics participants were inquired: development experience, attitude towards AmbientTalk, Eclipse IDE and debugging, and exceptions of a debugger for AmbientTalk programs.

Please answer the following questions with regard to your age, education and programming background. The answers will be kept private and only serve to put your other answers in context.

What is your age?:
Please sketch your educational background (Master, Bachelor, etc):

Please list the top 3 programming languages you are most comfortable with

## Questionnaire

For each of the statements below, please rate each one on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extend they represent your opinion.

| | |
|---|---|
| I consider myself an experienced developer | 1 2 3 4 5 |
| I am proficient in development distributed systems | 1 2 3 4 5 |
| I understand the principles of ambient oriented programming in AmbientTalk | 1 2 3 4 5 |
| Developing AmbientTalk is easy | 1 2 3 4 5 |
| I am familiar with the Eclipse IDE | 1 2 3 4 5 |
| Eclipse is a good IDE to develop AmbientTalk programs | 1 2 3 4 5 |
| A lot of bugs can be prevented by using better development tools | 1 2 3 4 5 |
| Debugging distributed programs is hard | 1 2 3 4 5 |
| Debuggers are a helpful tool to find errors in programs | 1 2 3 4 5 |
| Debuggers are a helpful tool to understand programs | 1 2 3 4 5 |
| I am proficient in the use of online (breakpoint-based) debuggers | 1 2 3 4 5 |
| Breakpoints are essential to debugging a program | 1 2 3 4 5 |
| Step-by-step execution is essential to debugging a program | 1 2 3 4 5 |
| Inspecting the program's state is essential to debugging a program | 1 2 3 4 5 |
| A debugger would make programming AmbientTalk easier | 1 2 3 4 5 |
| A debugger for AmbientTalk is needed | 1 2 3 4 5 |
| When debugging AmbientTalk programs messages between actors are more important than messages between objects | 1 2 3 4 5 |
| Inspecting mailbox contents is essential to debugging AmbientTalk programs | 1 2 3 4 5 |
| An AmbientTalk debugger must take into account the frequent disconnections and zero infrastructure phenomenon present in a mobile ad hoc networking setting | 1 2 3 4 5 |

## Comments

Please note what features would you like to have in a debugger for AmbientTalk, and what, if any, feature of the language might make it hard to develop AmbientTalk applications. (You can continue on the other side of the sheet.)

## B.2 Posttest Questionnaire

As previously mentioned, in the experiment conducted to evaluate REME-D , two questionnaires were used. This section prints the posttest questionnaire used to evaluate the experiment and gauge the participants' experiences with the tool. We first show the questions according to the different topics the participants were inquired. Afterwards, we include the questions in the original form used in the experiment. A more detailed explanation of each question's purpose is given in Section 11.2.2.

| | |
|---|---|
| **Assignment Experience** | |
| A | The assignment was too easy for me. |
| B | The assignment was very interesting to do. |
| C | The assignment represents the kind of bugs I have encountered in AmbientTalk. |
| D | I would have liked more time to complete the assignment. |
| E | I had enough help in completing the assignment. |
| **REME-D UI Experience** | |
| J | I found REME-D's user interface easy to use. |
| K | REME-D's debugging features are clear and accessible in the UI. |
| L | The Actor view gives a good overview of the state of the application. |
| M | The Debug Element View gives a good overview of the state of an actor. |
| N | In essence REME-D is helpful, but it needs a better user interface. |
| **REME-D Perception** | |
| F | REME-D makes AmbientTalk programming easier. |
| G | REME-D will significantly help to reduce the time to debug distributed AmbientTalk programs. |
| H | REME-D will help me understand the distributed behaviors of AmbientTalk programs. |
| I | REME-D will help me solve real bugs. |
| **Value of REME-D's features** | |
| P | I find asynchronous message breakpoints useful. |
| R | I find step-into a useful debugging operation. |
| T | I find step-over a useful debugging operation. |
| V | I find pausing an actor a useful debugging operation. |
| W | I find that REME-D's operations give me sufficient control over the execution of an AmbientTalk program. |
| X | I find that infecting other VM's is a useful feature when debugging distributed AmbientTalk programs. |
| **Frequency of REME-D's features** | |
| O | I often used asynchronous message-breakpoints during the assignment. |
| Q | I often used step-into during the assignment. |
| S | I often used step-over during the assignment. |
| U | I often paused an actor during the assignment. |

Table B.2: Posttest questions according to the five topics participants were inquired: assignment and REME-D UI experience, REME-D perception, value and frequency of REME-D's features.

# Questionnaire

For each of the statements below, please rate each one on a scale from 1 (totally disagree) to 5 (totally agree) to indicate to what extend they represent your opinion.

| | |
|---|---|
| The assignment was too easy for me | 1 2 3 4 5 |
| The assignment was very interesting to do | 1 2 3 4 5 |
| The assignment represents the kind of bugs I have encountered in AmbientTalk | 1 2 3 4 5 |
| I would have liked more time to complete the assignment | 1 2 3 4 5 |
| I had enough help in completing the assignment | 1 2 3 4 5 |
| REME-D makes AmbientTalk programming easier | 1 2 3 4 5 |
| REME-D will significantly help to reduce the time to debug distributed AmbientTalk programs | 1 2 3 4 5 |
| REME-D will help me understand the distributed behaviors of AmbientTalk programs | 1 2 3 4 5 |
| REME-D will help me solve real bugs | 1 2 3 4 5 |
| I found REME-D's user interface easy to use | 1 2 3 4 5 |
| REME-D's debugging features are clear and accessible in the UI | 1 2 3 4 5 |
| The Actor view gives a good overview of the state of the application | 1 2 3 4 5 |
| The Debug Element View gives a good overview of the state of an actor | 1 2 3 4 5 |
| In essence REME-D is helpful, but it needs a better user interface | 1 2 3 4 5 |
| I often used asynchronous message-breakpoints during the assignment | 1 2 3 4 5 |
| I find asynchronous message breakpoints useful | 1 2 3 4 5 |
| I often used step-into during the assignment | 1 2 3 4 5 |
| I find step-into a useful debugging operation | 1 2 3 4 5 |
| I often used step-over during the assignment | 1 2 3 4 5 |
| I find step-over a useful debugging operation | 1 2 3 4 5 |
| I often paused an actor during the assignment | 1 2 3 4 5 |
| I find pausing an actor a useful debugging operation | 1 2 3 4 5 |
| I find that REME-D's operations give me sufficient control over the execution of an AmbientTalk program | 1 2 3 4 5 |
| I find that infecting other VM's is a useful feature when debugging distributed AmbientTalk programs | 1 2 3 4 5 |

# Comments

Please note any additional comment you might have, either on the assignment, or on the tool. Please include features that you think might be enhanced, or added in order to make REME-D a better debugging tool (You can continue on the other side of the sheet).

## B.3 Debugging Assignment

A session assignment was handed out to participants in the experiment conducted to validate REME-D . The full text of the assignment is printed in this section in its original form. An description of the tasks within the assignment can be found in Section 11.2.3.

email: egonzale@vub.ac.be
office: 10F731

# goShopping: Debugging AmbientTalk programs with REME-D

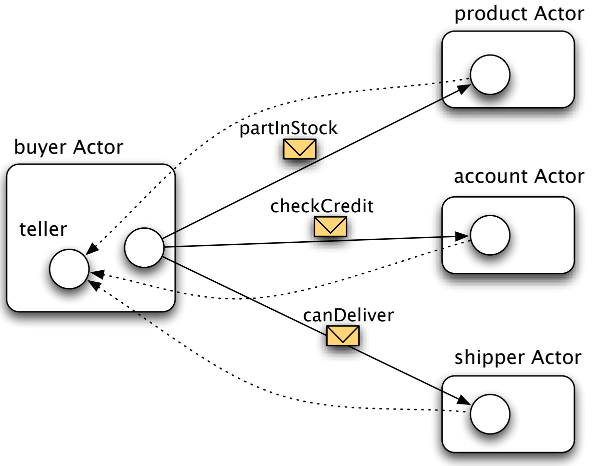
Lab session material available at Pointcarre under LabSessions, and at http://soft.vub.ac.be/~egonzale under Teaching.

## Idea

The purpose of this exercise is to get familiar with REME-D[a], a distributed debugger designed for AmbientTalk applications. To this end, the lab material provides you with an application that contains errors. You should try to fix them by launching it in the Eclipse AmbientTalk plugin in debug mode and using REME-D's features.

## Finding bugs in the goShopping application

The provided application is a sample shopping application that needs to process purchase orders. Before the shop can acknowledge the order, it must verify three things: 1) whether the requested items are still in stock, 2) whether the customer has provided valid payment information and 3) whether a shipper is available to ship the order in time. The following picture depicts this application which consists of 4 actors.



The buyer actor processes order purchases. In response to a go message, the buyer actor sends out the appropriate messages to product, account and shipper actors. To keep the first part of this exercise simple, we do not make use of futures. Instead, the buyer makes use of an *AsyncAnd* abstraction. The constructor of an AsyncAnd object takes two parameters: a number indicating how many affirmative replies the AsyncAnd should receive before it invokes callback<-run(true), and the callback object to notify. The callback object thus needs to implement the message `run(boolean)`. In the `goShopping` application provided, all three actors simply send an affirmative reply to the `AsyncAnd` callback.

Your task consists on fix and improve this application as follows:

(a) Running `goShopping.at` should print the following to the console "Got answer: true". However, it currently prints "Got answer: false" because the application does not behave as expected. Use REME-D debugging features to fix this bug.

(b) Once the shop acknowledges the order, it contacts a warranty broker to suggest the client a warranty for the purchases item. To do so, the warranty broker contacts several insurance agencies, and returns the best quote. However, it currently returns an negative quota. Use REME-D debugging features to fix this bug.

To reproduce the bug you need to 1) launch the warranty broker code (stored in the `warrantyBroker.at` provided in the lab session material), and 2) comment line 133 of `goShopping.at` to use instead the `goWithInsurance` ( sent using the `sectionB` method).

*Note*: In this case synchronization between buyer and the warranty broker happens by means of future-based message passing instead of using the explicit `AsyncAnd` abstraction.

(c) Add two unit test to the `goShopping` application to make sure these bugs do not happen again in the future.

[a] read as remedy

# Appendix C

# REME-D's Startup Protocol

This appendix demonstrates the interactions between debugger manager and command and event listeners by means of a concrete example. Figure C.1 gives a graphical overview of the startup protocol for a REME-D debugging session.

In response to a click on the debug button in the Eclipse IDE, the JavaVM running the Eclipse plugin creates two AmbientTalk VMs: the debugger VM and the target application VM. The debugger VM loads the code of the debugger actor shown in Figure 10.7 creating a debugger manager, and the command and event listeners. The target application VM loads the code of a local manager, which upon initialization announces its presence to the debugger manager by means of the `actorStarted` message. The debugger manager in turn communicates this to the event listener by sending a `startActorEvent` message. Finally, the listener notifies the corresponding Eclipse UI component (as shown in Figure 10.7, line 3) so that the debug view is updated with the information of the new actor.

Upon receiving the `startActorEvent` message, the Eclipse UI then creates a `startCommand` object including the breakpoints active on the UI. The execution of the `startCommand` object, notifies the command listener by invoking its `executeStartCommand` method. As shown in Figure 10.7 (lines 10-25), the command listener first obtains the breakpoint commands from the start command, and executes them. Executing a breakpoint command may either call the `codeBreakpointActiveOn` or `setCodeBreakpoint` method of the debugger manager as also shown in lines 17-23. Finally, the execution of the `executeStartCommand` method, sends a `loadMainCode` message to the debugger manager, which in turn sends an `evaluateCode` message to the local manager. In response to an `evaluateCode` message, the local manager loads the target application code, ending the debugging session startup protocol.

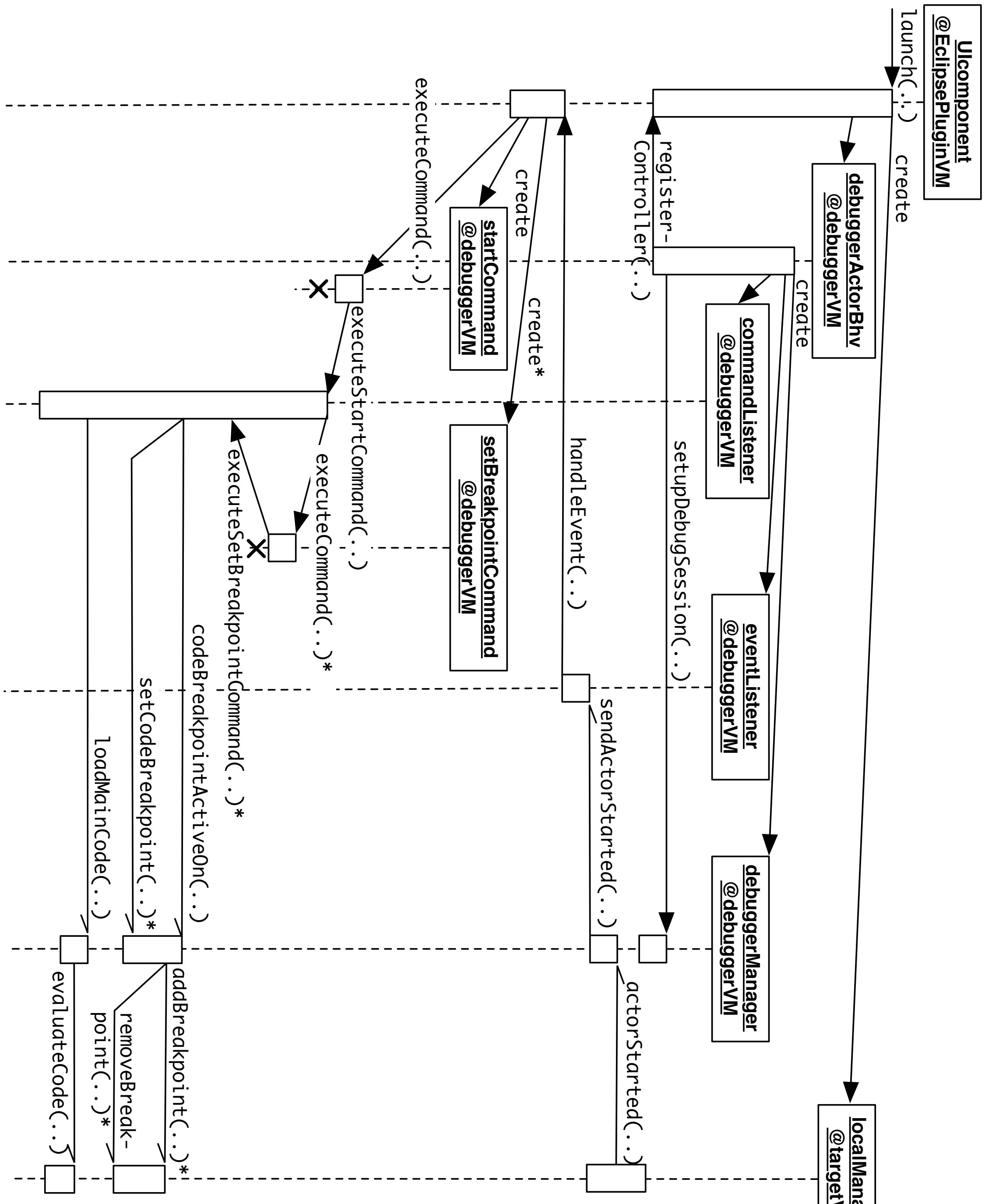


Figure C.1: REME-D's debugging session startup protocol.

# Bibliography

[A.08] Moore K. A. Quasi-experimental evaluations. part 6 in a series on practical evaluation methods. *Research-to-Results Brief*, 2008.

[ABC+00] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölszle, John Maloney, Randall Smith, David Ungar, and Mario Wolczko. The SELF 4.1 programmer's reference manual, 2000.

[ABW93] G. Nelson S. Owicki A. Birell, D. Evers and E. Wobber. Distributed garbage collection for network objects. Technical Report 116, Digital Systems Research Center, 1993.

[AC93] Gul Agha and C. J. Callsen. Actorspace: An open distributed programming paradigm. In *Proceedings of the 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices*, pages 23–32, 1993.

[ACDG98] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel reification: A reflective model for distributed computation. In *IEEE International Performance, Computing and Communications Conference (IPCCC'98)*, pages 32–36, 1998.

[ACH+08] Baruch Awerbuch, Reza Curtmola, David Holmer, Cristina Nita-Rotaru, and Herbert Rubens. Odsbr: An on-demand secure byzantine resilient routing protocol for wireless ad hoc networks. *ACM Trans. Inf. Syst. Secur.*, 10:6:1–6:35, January 2008.

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Transactions on aspect-oriented software development I. chapter An overview of caesarj, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[AK06] Markus Aleksy and Axel Korthaus. Using temporary properties to provide leasing functionality in corba. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 01*, AINA '06, pages 941–946, Washington, DC, USA, 2006. IEEE Computer Society.

[AKS05] Markus Aleksy, Axel Korthaus, and Martin Schader. Realizing the leasing concept in corba-based applications. In *Proc. of Symp. on Applied Comp.*, pages 706–712, 2005.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[Alv09] Nicolas Cardozo Alvarez. Parameter passing semantics for mobile adhoc networks. Master's thesis, Vrije Universiteit Brussel in collaboration with Ecole des Mines de Nantes, August 2009.

[AR98] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. In *ACM Computing Surveys*, volume 30, pages 330–373, 1998.

[Ast12] Patricio Javier Astudillo. A distributed back-in-time debugger for ambient-oriented programs. Master's thesis, Vrije Universiteit Brussels, Faculty of Sciences, Software Languages Lab, 2012.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.

[BBC$^+$01] Françoise Baude, Alexandre Bergel, Denis Caromel, Fabrice Huet, Olivier Nano, and Julien Vayssière. Ic2d: Interactive control and debugging of distribution. In *Proceedings of the Third International Conference on Large-Scale Scientific Computing-Revised Papers*, LSSC '01, pages 193–200, London, UK, UK, 2001. Springer-Verlag.

[BC06] Paolo Bellavista and Antonio Corradi. *The Handbook of Mobile Middleware*. Auerbach Publications, Boston, MA, USA, 2006.

[BCA$^+$01] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The design and implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2:–, June 2001.

[BDG02] Romain Boichat, Partha Dutta, and Rachid Guerraoui. Asynchronous leasing. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, WORDS '02, pages 180–187, Washington, DC, USA, 2002. IEEE Computer Society.

[Ben86] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986.

[Bev87] David I. Bevan. Distributed garbage collection using reference counting. In *Parlallel Architectures and Languages Europe*, pages 176–187. Springer-Verlag, 1987.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a Production Environment. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 215–230, 1993.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[Bis77] P. B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977.

[BMR03] Kevin Bowers, Kevin Mills, and Scott Rose. Self-adaptive leasing for jini. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 539, Washington, DC, USA, 2003. IEEE Computer Society.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[BNOW93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 217–230, New York, NY, USA, 1993. ACM.

[BP06] Hoimonti Basu and Jan Pedersen. Idli: An interactive message debugger for parallel programs using lam-mpi. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications Conference on Real-Time Computing Systems and Applications, PDPTA 2006*, volume 1, pages 513–520, Las Vegas, Nevada, USA, June 2006. CSREA Press.

[BRL01] Randal C. Burns, Robert M. Rees, and Darell D. E. Long. An analytical study of opportunistic lease renewal. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 146–153, Washington, DC, USA, 2001. IEEE Computer Society.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming languages for distributed computing systems. *ACM Comput. Surv.*, 21(3):261–322, 1989.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[BVB$^+$12] Engineer Bainomugisha, Jorge Vallejos, Elisa Gonzalez Boix, Pascal Costanza, Theo D'Hondt, and Wolfgang De Meuter. Bringing scheme programming to the iPhone Experience. *Software: Practice and Experience*, 42(3):331–356, 2012.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 286–297. ACM Press, 1995.

[Caz01] Walter Cazzola. *Communication-Oriented Reflection: a Way to Open Up the RMI Mechanism*. PhD thesis, Department of Computer Science, Universita degli Studi di Milano, February 2001.

[CBM90] Wing Hong Cheung, James P. Black, and Eric Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, 1990.

[CC03] Alvin T. S. Chan and Siu-Nam Chuang. Mobipads: A reflective middleware for context-aware mobile computing. *IEEE Trans. Softw. Eng.*, 29(12):1072–1085, December 2003.

[CCC02] Siu-Nam Chuang, Alvin T. S. Chan, and Jiannong Cao. Dynamic service composition for wireless web access. In *Proceedings of the 2002 International Conference on Parallel Processing*, ICPP '02, pages 429–, Washington, DC, USA, 2002. IEEE Computer Society.

[CDDS94] Wim Codenie, Koen D'Hont, Theo D'Hondt, and Patrick Steyaert. Agora: Message passing as a foundation for exploring OO language concepts. *SIGPLAN Notices*, 29(12):48–57, 1994.

[CDK05] Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[CDNF01] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Trans. Softw. Eng.*, 27(9):827–850, September 2001.

[CEM03] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Trans. Softw. Eng.*, 29(10):929–945, October 2003.

[CGS$^+$05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.

[CHV01] Denis Caromel, Fabrice Huet, and Julien Vayssiere. A simple security-aware mop for java. In *International Conference in Metalevel Architectures and Separation of Concerns (Reflection 2001)*, volume 2192, pages 118–125. Springer-Verlag, 2001.

[CJ02] Gianpaolo Cugola and H.-Arno Jacobsen. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):25–33, 2002.

[CKV98] Denis Caromel, Wilfried Klauser, and Julien Vayssire. Towards seamless computing and metacomputing in java. *Concurrency: Practice and Experience*, 10(11-13):1043–1061, 1998.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *In 5th International Symposium on Formal Methods for Components and Objects (FMCO*. Springer-Verlag, 2006.

[CMM09] Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Linguistic symbiosis between event loop actors and threads. *Computer Languages, Systems Structures*, 35(1):80 – 98, 2009.

[Col07] Raphael Collet. *The Limits of Network Transparency in a Distributed Programming Language*. PhD thesis, Faculte des Sciences Appliquees, Departement d'Ingenierie Informatique, Universite Catholique de Louvain, December 2007.

[CPN98] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[CR06] Raphael Collet and Peter Van Roy. Failure handling in a network-transparent distributed programming language. *Advanced Topics in Exception Handling Techniques*, pages 121–140, 2006.

[CRW00] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 219–227, New York, NY, USA, 2000. ACM.

[CS63] D.T. Campbell and J.C. Stanley. *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Company, 1963.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[CV01] Denis Caromel and Julien Vayssire. Reflections on mops, components, and java security. In *ECOOP*, pages 256–274. Springer-Verlag, 2001.

[CWÖJ08] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *APLAS*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2008.

[DA07] Bill Donkervoet and Gul Agha. Reflecting on aspect-oriented programming, metaprogramming, and adaptive distributed monitoring. In *Proceedings of the 5th international conference on Formal methods for components and objects*, FMCO'06, pages 246–265, Berlin, Heidelberg, 2007. Springer-Verlag.

[DAKV09] Darren Dao, Jeannie Albrecht, Charles Killian, and Amin Vahdat. Live debugging of distributed systems. In *Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, CC '09, pages 94–108, Berlin, Heidelberg, 2009. Springer-Verlag.

[DBT$^+$06] Adam Drozd, Steve Benford, Nick Tandavanitj, Michael Wright, and Alan Chamberlain. Hitchers: Designing for cellular positioning. In *Ubicomp*, pages 279–296, 2006.

[DDD04] Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker. Pico: Scheme for mere mortals. 2004.

[DDT12] Allinea DDT. The distributed debugging tool. Technical report, Allinea, 2012. `http://www.allinea.com/Portals/90122/docs/user-guides-and-technical-docs/allinea-ddt-3.1-user-guide-may-2012.pdf` (captured May 2012).

[Ded06] Jessie Dedecker. *Ambient-Oriented Programming*. PhD thesis, Faculteit Wetenschappen, Vrije Universiteit Brussel, May 2006.

[DFWB98] Nigel Davies, Adrian Friday, Stephen P. Wade, and Gordon S. Blair. L2imbo: a distributed systems platform for mobile computing. *Mob. Netw. Appl.*, 3(2):143–156, 1998.

[DGG02] Assia Doudou, Benot Garbinato, and Rachid Guerraoui. Encapsulating failure detection: from crash to byzantine failures. In *Proc. International Conference on Reliable Software Technologies*, pages 24–50. Springer-Verlag, 2002.

[DGM$^+$07] Jessie Dedecker, Elisa Gonzalez Boix, Stijn Mostinckx, Stijn Timbermont, Jorge Vallejos, and Tom Van Cutsem. The AmbientTalk/2 tutorial. 2007. `http://soft.vub.ac.be/amop/at/tutorial/tutorial` (captured in September 2011).

[D'H96] Theo D'Hondt. The pico programming project. 1996. `http://pico.vub.ac.be` (captured in September 2011).

[DMQ07] C. Dabrowski, K. Mills, and S. Quirolgico. Understanding failure response in service discovery systems. *The Journal of Systems and Software*, 80:896–917, June 2007.

[DNO98] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *Proceedings of the 1998 ACM symposium on Applied Computing*, SAC '98, pages 169–177, New York, NY, USA, 1998. ACM.

[DS10] Magdalena Dukielska and Jacek Sroka. Javaspaces netbeans: a linda workbench for distributed programming course. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 23–27, New York, NY, USA, 2010. ACM.

[DST03] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1266–1276, September 2003.

[DVM$^+$06] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer, 2006.

[DWZVD09] Michiel De Wit, Andy Zaidman, and Arie Van Deursen. Managing code clones using dynamic change tracking and resolution. In *IEEE International Conference on Software Maintenance (ICSM 2009)*, pages 169–178. IEEE, September 2009.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34:375–408, September 2002.

[EFGA03] P. Th. Eugster, Pascal A. Felber, R. Guerraoui, and A.Kermarrec. The many faces of publish/subscribe. *ACM Computing Survey*, 35(2):114–131, 2003.

[EGH05] P.Th. Eugster, B. Garbinato, and A. Holzer. Location-based publish/-subscribe. *Fourth IEEE International Symposium on Network Computing and Applications*, pages 279–282, 2005.

[EGH06] Patrick Eugster, Benoit Garbinato, and Adrian Holzer. Pervaho: A development & test platform for mobile ad hoc applications. In *Third annual International Conference on Mobile and Ubiquitous Systems: Networking & Services*, pages 1–5, July 2006.

[Els89] I. J. P. Elshoff. A distributed debugger for amoeba. *SIGPLAN Not.*, 24(1):1–10, 1989.

[ET01] John Eberhard and Anand Tripathi. Efficient object caching for distributed java rmi applications. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 15–35, London, UK, 2001. Springer-Verlag.

[Eug03] Patrick Th. Eugster. Lazy Parameter Passing. Technical report, Sun Microsystems, 2003.

[Eug06] Patrick Eugster. Uniform proxies for java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM.

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, 1999.

[FGF99] Pascal Felber, Rachid Guerraoui, and Mohamed E. Fayad. Putting oo distributed programming to work. *Commun. ACM*, 42:97–101, November 1999.

[FHL98] Michael Frumkin, Robert Hood, and Louis Lopez. Trace-driven debugging of message passing programs. In *12th International Parallel Processing Symposium*, IPPS '98, pages 753–762, Washington, DC, USA, 1998. IEEE Computer Society.

[For89] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence & Databases*, pages 547–557. Kaufmann Publishers, INC., San Mateo, CA, 1989.

[FPK$^+$07] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation*, pages 271 – 284, Cambridge MA, USA, April 2007.

[FR07] D. Frey and G-C. Roman. Context-aware publish subscribe in mobile ad hoc networks. In *9th International Conference on Coordination Models and Languages (COORDINATION)*, volume 4467 of *LNCS*, pages 37–55. Springer, June 2007.

[Gai85] Jason Gait. A debugger for concurrent programs. *Software: Practice and Experience*, 15(6):539–554, 1985.

[Gar05] Jesse James Garrett. Ajax: A new approach to web applications. `http://adaptivepath.com/ideas/essays/archives/000385.php`, February 2005. [Online; Stand 18.03.2008].

[GBCVC$^+$11] E. Gonzalez Boix, Noguera C., T. Van Cutsem, W. De Meuter, and T. D'Hondt. REME-D: a Reflective, Epidemic Message-Oriented Debugger for Ambient-Oriented Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011*, volume 2, pages 1275–1281. ACM, 2011.

[GBLCS$^+$11] E. Gonzalez Boix, A. Lombide Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, and T. D'Hondt. Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011*, volume 1, pages 425–432. ACM, 2011.

[GBS03] Paul Grace, Gordon S. Blair, and Sam Samuel. Remmoc: A reflective middleware to support mobile client interoperability. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1170–1187, Catania, Sicily, Italy, November 2003. Springer.

[GBS05] Paul Grace, Gordon S. Blair, and Sam Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, January 2005.

[GBVCJ+09] E. Gonzalez Boix, T. Van Cutsem, Vallejos J., W. De Meuter, and T. D'Hondt. A Leasing Model to Deal with Partial Failures in Mobile Ad Hoc Networks. In *In the 47th International Conference on Objects, Models, Components, Patterns (TOOLS 2009)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 231–251. Springer-Verlag Berlin Heidelberg, June 2009.

[GC89] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[GCG01] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th annual ACM symposium on Principles of distributed computing*, PODC '01, pages 170–179, New York, NY, USA, 2001. ACM.

[GDL+04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan 1985.

[GF99] R. Guerraoui and M. E. Fayad. OO Distributed Programming is *Not* Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, 1999.

[GGM94] Benoît Garbinato, Rachid Guerraoui, and Karim Mazouni. Distributed programming in garf. In *Proceedings of the Workshop on Object-Based Distributed Programming*, pages 225–239, London, UK, 1994. Springer-Verlag.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: a holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[Gol89] Benjamin Goldberg. Generational reference counting: A reduced communication distributed storage reclamation scheme. In ACM SIGPLAN, editor, *Programming Languages Design and Implementation*, volume 24, pages 313–321, 1989.

[Got09] Chris Gottbrath. Deterministically troubleshooting network applications. Technical report, TotalView Technologies, april 2009.

[GPP+10] Wojciech Galuba, Panos Papadimitratos, Marcin Poturalski, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Castor: Scalable secure routing for ad-hoc networks. In *IEEE INFOCOM*, pages 1–9. IEEE Computer Society, 2010.

[GR06] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Gra04] Paul Grace. *Overcoming Middleware Heterogeneity in Mobile Computing Applications*. PhD thesis, Computing Department, Lancaster University, March 2004.

[GSL+10] Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Scripting mobile devices with ambienttalk. *Handheld Computing for Mobile Commerce: Applications, Concepts and Technologies*, pages 202–224, 2010.

[GTKT08] Sriram Gopal, Wesley Tansey, Gokulnath C. Kannan, and Eli Tilevich. Dexter: an extensible framework for declarative parameter passing in distributed object systems. In *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 144–163, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[GVV+07] Elisa Gonzalez Boix, Jorge Vallejos Vargas, Tom Van Cutsem, Jessie Dedecker, and Wolfgang De Meuter. Context-aware leasing for mobile ad hoc networks. In *3rd Workshop on Object-Oriented technology for Ambient Intelligence and Pervasive Computing (OT4AmI) co-located at the European Conference on Object-Oriented Programming (ECOOP'07)*, 2007.

[HGDD12] Dries Harnie, Elisa Gonzalez Boix, Wolfgang De Meuter, and Theo D'Hondt. Programming urban-area applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 1516–1521. ACM, 2012.

[HGM04] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10(6):643–652, November 2004.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '93, pages 411–428, New York, NY, USA, 1993. ACM.

[Hoo96] Robert Hood. The p2d2 project: building a portable distributed debugger. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools (SPDT)*, pages 127–136, New York, NY, USA, 1996. ACM.

[HRBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report 94-1425, Computer Science Department, Cornell University, May 1994.

[HY88] Yasuaki Honda and Akinori Yonezawa. Debugging concurrent systems based on object groups. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 267–282, London, UK, 1988. Springer-Verlag.

[IA06] A. Iliasov and Romanovsky A. Structured coordination spaces for fault tolerant mobile agents. *Advanced Topics in Exception Handling Techniques*, 4119:181–199, 2006.

[JdT$^+$95] Anthony D. Joseph, Alan F. deLespinasse, Joshua A. Tauber, David K. Gifford, and M. Frans Kaashoek. Rover: a toolkit for mobile information access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 156–171, Colorado, December 1995.

[JHS$^+$10] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proceedings of the 9th international conference on Formal Methods for Components and Objects*, volume 6957 of *FMCO'10*, pages 142–164. Springer-Verlag, 2010.

[jin03] The jini distributed leasing specification version 1.0, 2003. `http://river.apache.org/doc/specs/html/lease-spec.html`.

[JJ92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *In Proc. IWMM, volume 637 of Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, 1992.

[JK00] Prashant Jain and Michael Kircher. Leasing. In *Proceedings of the 7th Patterns Languages of Programs Conference (PLoP)*, pages –, 2000.

[JL93] R. Jones and R. Lins. Cyclic weighted reference counting without delay. In *Proceedings of Parlallel Architectures and Languages Europe*, pages 712–715. Springer-Verlang, 1993.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JLZ11] Einar Broch Johnsen, Ivan Lanese, and Gianluigi Zavattaro. Fault in the future. In *Proceedings of the 13th international conference on Coordination models and languages*, COORDINATION'11, pages 1–15, Berlin, Heidelberg, 2011. Springer-Verlag.

[jnd03] Java naming and directory interface version 1.2. Technical report, Sun Microsystems, Inc., 2003.

[JO07] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, mar 2007.

[JR04] Christine Julien and Gruia-Catalin Roman. Active coordination in ad hoc networks. In Rocco De Nicola, Gian Luigi Ferrari, and Greg Meredith, editors, *Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings*, volume 2949 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2004.

[KB02] Alan Kaminsky and Hans-Peter Bischof. Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In *ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 72–73. ACM Press, 2002.

[KDHKS09] Vigdis By Kampenes, Tore Dybå, Jo E. Hannay, and Dag I. K. Sjøberg. A systematic review of quasi-experiments in software engineering. *Inf. Softw. Technol.*, 51(1):71–82, January 2009.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. Technical report, Los Alamitos, CA, USA, 1996.

[Kie96] Thilo Kielmann. Designing a coordination model for open systems. In *Proceedings of the First International Conference on Coordination Languages and Models*, COORDINATION '96, pages 267–284, London, UK, UK, 1996. Springer-Verlag.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[KO96] Bent Bruun Kristensen and Kasper . Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2:143–160, December 1996.

[KRL$^+$00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalhã, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *IFIP/ACM International Conference on Distributed systems platforms*, Middleware '00, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, February 1992.

[KSMA06] Youngmin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: An actor platform for wireless sensor networks. In *In Proc.*

*of the 5th International Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, AAMAS '06, pages 1297–1300, New York, NY, USA, 2006. ACM.

[KTA09] Young-Woo Kwon, Eli Tilevich, and Taweesup Apiwattanapong. DR-OSGi: hardening distributed components with network volatility resiliency. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, pages 19:1–19:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21(7):558–565, 1978.

[Lam87] Leslie Lamport. Distribution e-mail, May 1987. `http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt` (captured in September 2011).

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of argus. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 111–122, New York, NY, USA, 1987. ACM.

[Led99] Thomas Ledoux. Opencorba: a reflective open broker. In *Reflection'99*, volume 1616 of *LNCS*, pages 197–214, Saint-Malo, France, July 1999.

[Lew03] Bil Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.

[Lin92] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications Of The ACM*, 31(3):300–312, 1988.

[LJE08] Christopher Line, K. R. Jayaram, and Patrick Eugster. Lazy argument passing in java rmi. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 127–136, New York, NY, USA, 2008. ACM.

[LL86] Barbara Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th ACM Symp. Principles of Distributed Computing*, pages 29–39, 1986.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987.

[LMMB02] Fiege Ludger, Mira Mezini, Gero Mühl, and Alejandro P. Buchmann. Engineering event-based systems with scopes. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 309–333, London, UK, 2002. Springer-Verlag.

[Lom11] Andoni Lombide Carreton. *Ambient-Oriented Dataflow Programming for Mobile RFID-Enabled Applications*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab, October 2011.

[Lop96] Cristina Videira Lopes. Adaptive parameter passing. In *In proceedings of the second International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, pages 1 – 25, 1996.

[Low03] Juval Lowy. Managing the lifetime of remote .net objects with leasing and sponsorship. *MSDN Library*, December 2003.

[LQP92] Bernard Lang, Christian Queinnec, and Jose Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 39–50, 1992.

[LS88] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[MC02] René Meier and Vinny Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *22nd International Conference on Distributed Computing Systems*, pages 639–644, Washington, DC, USA, 2002. IEEE Computer Society.

[MC03] R. Meier and V. Cahill. Exploiting proximity in event-based middleware for collaborative mobile applications. In *Distributed Applications and Interoperable Systems*, 2003.

[McA95] Jeff McAffer. Meta-level programming with coda. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214, London, UK, 1995. Springer-Verlag.

[MCC$^+$95] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce, Irvin Karen, L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 1995.

[MDG$^+$06] S. Mostinckx, Jessie Dedecker, Elisa Gonzalez Boix, Tom Van Cutsem, and Wolfgang De Meuter. Ambient-Oriented Exception Handling. *Advanced Topics in Exception Handling Techniques*, 2006.

[ME03] Gareth P. McSorley and Huw Evans. Tiamat: Generative communication in a changing world. In *Middleware Workshops*, pages 37–44. PUC-Rio, 2003.

[Mei02] René Meier. Communication paradigms for mobile computing. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):56–58, 2002.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[MH89] Charles E. Mcdowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21:593–622, 1989.

[MHS02] R. Sharma J. Fialli M. Hapner, R. Burridge and K. Stout. Java message service specification version 1.1. Technical report, Sun Microsystems, Inc., 2002.

[Mil06] M. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, John Hopkins University, Baltimore, Maryland, USA, May 2006.

[ML95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, 1995.

[MLE02] C. Mascolo, L.Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag, 2002.

[MMF00] Mark Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proceedings of the 4th International Conference on Financial Cryptography*, FC '00, pages 349–378, London, UK, UK, 2000. Springer-Verlag.

[MMH05] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. Emma: Epidemic messaging middleware for ad hoc networks. *Personal Ubiquitous Comput.*, 10(1):28–36, 2005.

[MMP$^+$96] Michael S. Meier, Kevan L. Miller, Donald P. Pazel, Josyula R. Rao, and James R. Russell. Experiences with building distributed debuggers. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '96, pages 70–79, New York, NY, USA, 1996. ACM.

[MP06] Amy L. Murphy and G.P Picco. Using lime to support replication for availability in mobile ad hoc networks. In *8th International Conference on Coordination Models and Languages (COORDINATION)*, LNCS, pages 194–211. Springer-Verlag, 2006.

[MPR01] Amy L. Murphy, G. Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, pages 524–536. IEEE Computer Society, 2001.

[MRM06] Marija Mikic-Rakic and Nenad Medvidovic. A classification of disconnected operation techniques. *EUROMICRO Conference*, 0:144–151, 2006.

[MRV98] A. L. Murphy, G.-C. Roman, and G. Varghese. An exercise in formal reasoning about mobile communications. In *IWSSD '98: Proceedings of the 9th international workshop on Software specification and design*, page 25, Washington, DC, USA, 1998. IEEE Computer Society.

[MS03] Mark Miller and Jonathan Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Proceedings of Eighth Asian Computing Science Conference*, ASIAN'03, pages 224–242, December 2003.

[MTS05] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005.

[MVT+09] Stijn Mostinckx, Tom Van Cutsem, Stijn Timbermont, Elisa Gonzalez Boix, Éric Tanter, and Wolfgang De Meuter. Mirror-based reflection in AmbientTalk. *Software: Practice and Experience*, 39(7):661–699, 2009.

[MW00] Iain Merrick and Alan Wood. Coordination with scopes. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 210–217, New York, NY, USA, 2000. ACM.

[MWN02] Scott McLean, Kim Williams, and James Naftel. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002.

[MZ04] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications*, pages 263–273, Washington, DC, USA, 2004. IEEE Computer Society.

[MZ09] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):15:1–15:56, July 2009.

[MZS+10] Nick Matthijssen, Andy Zaidman, Margaret-Anne Storey, Ian Bull, and Arie van Deursen. Connecting traces: Understanding client-server interactions in ajax applications. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 216–225, Washington, DC, USA, 2010. IEEE Computer Society.

[Nak09] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2009. `http://fastbull.dl.sourceforge.net/project/bitcoin/Design%20Paper/bitcoin.pdf/bitcoin.pdf` (captured in August 2012).

[Nik00] Pekka Nikander. Fault tolerance in decentralized and loosely coupled systems. In *Proceedings of Ericsson Conference on Software Engineering (ECSE'2000)*, September 2000.

[NKS$^+$02] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative leases: scalable consistency maintenance in content distribution networks. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 1–12, New York, NY, USA, 2002. ACM.

[NKSI05] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *SIGPLAN Not.*, 40(6):249–260, 2005.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 502–511, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[NNE04] Panayiotis Neophytou, Neophytos Neophytou, and Paraskevas Evripidou. Debugging MPI grid applications using Net-dbx. In *European Across Grids Conference*, Lecture Notes in Computer Science, pages 139–148, 2004.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[ope] Visualworks Opentalk Developer's Guide - Part Number: P46-0135-06.

[ORV05] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Time-aware coordination in respect. In *Proceedings of the 7th international conference on Coordination Models and Languages*, COORDINATION'05, pages 268–282, Berlin, Heidelberg, 2005. Springer-Verlag.

[OZ99] Andrea Omicini and Franco Zambonelli. Tuple centres for the coordination of Internet agents. In *1999 ACM Symposium on Applied Computing (SAC'99)*, pages 183–190, San Antonio, TX, USA, February 1999. ACM. Special Track on Coordination Models, Languages and Applications.

[PA98] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

[PCBB07] Julien Pauty, Paul Couderc, Michel Banatre, and Yolande Berbers. Geo-linda: a geometry aware distributed tuple space. In *AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, pages 370–377, Washington, DC, USA, 2007. IEEE Computer Society.

[Per90] Mark Perlin. Scaffolding the RETE network. In *International Conference on Tools for Artificial Intelligence*, pages 378–385. IEEE Computer Society, 1990.

[PHD11] Kevin Pinte, Dries Harnie, and Theo D'Hondt. Enabling cross-technology mobile applications with network-aware references. In *13th International Conference on Coordination Models and Languages, COORDINATION 2011*, volume 6721 of *LNCS*, Heidelberg, 2011. Springer-Verlag.

[PHGD11] Kevin Pinte, Dries Harnie, Elisa Gonzalez Boix, and Wolfgang De Meuter. Network-aware references for pervasive social applications. In *Second IEEE Workshop on Pervasive Collaboration and Social Networking (PERCOM workshops)*, pages 537–542, 2011.

[PHM06] Panagiotis Papadimitratos, Zygmunt J. Haas, and Senior Member. Secure data communication in mobile ad hoc networks. *IEEE Journal On Selected Areas In Communications*, 24:343–356, 2006.

[Piq91] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. In E. Aarts and J. van Leeuwen, editors, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE'91), Eindhoven, The Netherlands*, volume 505 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, June 1991.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. Lime: Linda meets mobility. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 368–377, New York, NY, USA, 1999. ACM.

[PN93] Cherri M. Pancake and Robert H. B. Netzer. A bibliography of parallel debuggers, 1993 edition. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*, PADD '93, pages 169–186, New York, NY, USA, 1993. ACM.

[PN04] Cherri M. Pancake and Robert H. B. Netzer. Bibliography on parallel and distributed debuggers, 2004. `http://liinwww.ira.uka.de/bibliography/Parallel/debug_3.1.html` (captured in June 2012).

[PS95] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Kinross, Scotland (UK), 1995.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 535–552, New York, NY, USA, 2007. ACM.

[Qui07] John Quigley. The white programming language, 2007. CODE Group, Illinois Institute of Technology. `http://dijkstra.cs.iit.edu/code/white`.

[RI04] Manuel Roman and Nayeem Islam. Dynamically programmable and reconfigurable middleware services. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware

'04, pages 372–396, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[RJ98] Helena Rodrigues and Richard Jones. Cyclic distributed garbage collection with group merger. In *ECOOP '98: Proceedings of the 12th European conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 260–273, 1998.

[RK98a] Thomas Riechmann and Jrgen Kleinder. Meta objects for access control: Role-based principals. In Colin Boyd and Ed Dawson, editors, *ACISP*, volume 1438 of *Lecture Notes in Computer Science*, pages 296–307. Springer, 1998.

[RK98b] M RONSSE and D KRANZLMULLER. Rolt(mp) - replay of lamport timestamps for message passing systems. *PROCEEDINGS OF THE SIXTH EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING - PDP '98*, pages 87–93, 1998.

[RKC01] Manuel Roman, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online*, 2(5), 2001.

[Roy99] Peter Van Roy. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in mozart. In *In International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99)*, July 1999.

[SAB$^+$94] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. A scalable debugger for massively parallel message-passing programs. *IEEE Parallel Distrib. Technol.*, 2(2):50–56, June 1994.

[SBBK95] A. Schill, B. Bellmann, W. Bohmak, and S. Kummel. System support for mobile distributed applications. In *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments*, SDNE '95, pages 124–, Washington, DC, USA, 1995. IEEE Computer Society.

[SCM09] Terry Stanley, Tyler Close, and Mark S. Miller. Causeway: A message-oriented distributed debugger. Technical Report HPL-2009-78, HP Laboratories, 2009.

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust distributed references and acyclic garbage collection. In *Proceedings of the 11th Symposium on Principles of Distributed Computing (PODC)*, PODC '92, pages 135–146, New York, NY, USA, 1992. ACM.

[SGBDMD10] C. Scholliers, E. Gonzalez Boix, W. De Meuter, and T. D'Hondt. Context-aware tuples for the ambient. In *On the Move to Meaningful Internet Systems, OTM 2010*, pages 745–763. Springer, 2010.

[SGD09] Christophe Scholliers, Elisa Gonzalez Boix, and Wolfgang De Meuter. Totam: Scoped tuples for the ambient. In *Proc. of the CAMPUS Workshop collocated with DisCoTec'09 federated event*, volume 19, pages 19–34. EASST, 2009.

[SGL06] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *OOPSLA Companion*, pages 975–985, New York, NY, USA, 2006. ACM Press.

[SJ07] Drew Stovall and Christine Julien. Resource discovery with evolving tuples. In *ESSPE '07: International workshop on Engineering of software services for pervasive environments*, pages 1–10, New York, NY, USA, 2007. ACM.

[Smi84] Brian Cantwell Smith. Reflection and semantics in LISP. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM Press.

[SP07] Christophe Scholliers and Eline Philips. Coordination in volatile networks. Master's thesis, Vrije Universiteit Brussels, 2007.

[SPG90] Marc Shapiro, David Plainfoss, and Olivier Gruber. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA-Roquencourt, 1990.

[Sri06] Nigamanth Sridhar. Decentralized local failure detection in dynamic distributed systems. *Reliable Distributed Systems, IEEE Symposium on*, 0:143–154, 2006.

[Sun98] Sun Microsystems. Java RMI specification, 1998. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[Tan08] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 168–179, New York, NY, USA, 2008. ACM.

[TKV02] Nam Thoai, Dieter Kranzlmüller, and Jens Volkert. Shortcut replay: A replay technique for debugging long-running parallel programs. In *Proceedings of the7th Asian Computing Science Conference on Advances in Computing Science: Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster*, ASIAN '02, pages 34–46, London, UK, UK, 2002. Springer-Verlag.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, pages 27–46. ACM, 2003.

[TP05] Erik Tribou and Jan Pedersen. Millipede: A multilevel debugging environment for distributed systems. In *Proc. of the Inter. Conf. on Parallel and Distributed Processing Techniques and Appl. (PDPTA)*, volume 1, pages 187–193, Las Vegas Nevada, USA, 2005.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[TS03] Eli Tilevich and Yannis Smaragdakis. Nrmi: Natural and efficient middleware. In *ICDCS'03*, pages 252–252, 2003.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Val11] Jorge Vallejos. *Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab, July 2011.

[Van08] Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad hoc Networks*. PhD thesis, Faculteit Wetenschappen, Programming Technology Lab, May 2008.

[VC09] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In *COORDINATION '09: Proceedings of the 11th International Conference on Coordination Models and Languages*, pages 143–162, Berlin, Heidelberg, 2009. Springer-Verlag.

[VF05] Luis Veiga and Paulo Ferreira. Asynchronous complete distributed garbage collection. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, pages –. IEEE Computer Society, 2005.

[VMG$^+$07] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society (SCCC)*, pages 3–12. IEEE Computer Society, 2007.

[VO06] M. Viroli and A. Omicini. Coordination as a service. *Fundamenta Informaticae*, 73(4):507–534, 2006.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52:40–44, January 2009.

[VT95] Nalini Venkatasubramanian and Carolyn Talcott. Reasoning about meta level activities in open distributed systems. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed* *computing*, PODC '95, pages 144–152, New York, NY, USA, 1995. ACM.

[VVG$^+$07] Jorge Vallejos Vargas, Tom Van Cutsem, Elisa Gonzalez Boix, Stijn Mostinckx, Jessie Dedecker, and Wolfgang De Meuter. The message-oriented mobility model. *Special Issue on Journal of Object Technology (JOT)*, 6(9):363–382, 2007.

[VZKS11] Roman Vitenberg, Dmitry Zinenko, Kristian Kvilekval, and Ambuj Singh. Analyzing performance of lease-based schemes under failures. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, pages 193–202, Washington, DC, USA, 2011. IEEE Computer Society.

[Wal99] Jim Waldo. The Jini Architecture for Network-centric Computing. *Commun. ACM*, 42(7):76–82, 1999.

[Wal01] Jim Waldo. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)*, pages 9–20, 2001.

[WCS02] Xingfu Wu, Qingping Chen, and Xian-He Sun. Design and development of a scalable distributed debugger for cluster computing. *Cluster Computing*, 5(4):365–375, October 2002.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[WI87] P. Watson and I.Watson. An efficient garbage collection scheme for parallel computer architecture. In *Parallel Architectures and Languages Europe*, pages 432–443. Springer-Verlang, 1987.

[Wis97] Roland Wismüller. Debugging message passing programs using invisible message tags. In *Proc. of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 295–302. Springer-Verlag, 1997.

[WMLF98] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, July 1998.

[WS99] Ian Welch and Robert J. Stroud. From dalang to kava - the evolution of a reflective java extension. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, Reflection '99, pages 2–21, London, UK, 1999. Springer-Verlag.

[WS00] Ian Welch and Robert J. Stroud. Kava - a reflective java based on bytecode rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999*, pages 155–167, London, UK, 2000. Springer-Verlag.

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object*

*Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.

[YADL99] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):563–576, July 1999.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.

[YCC$^+$06] Chih-Chieh Yang, Chung-Kai Chen, Yu-Hao Chang, Kai-Hsin Chung, and Jenq-Kuen Lee. Streaming support for java rmi in distributed environments. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 53–61, New York, NY, USA, 2006. ACM.

[ZGSK05] Shelley Q. Zhuang, Dennis Geels, Ion Stoica, and Randy H. Katz. On failure detection algorithms in overlay networks. In *IN IEEE INFOCOM*, pages 2112–2123, 2005.