Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with Ecole des Mines de Nantes - France

2005

Mobile Actors Supporting Reconfigurable Applications in Open Peer-to-Peer Networks

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Jorge Vallejos Vargas

Promotor: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Advisors: Tom Van Cutsem and Stijn Mostinckx (Vrije Universiteit Brussel)

Abstract

Mobile technology is gently beginning to seep into society. With it, new visions of computing can be realized, where users are continually surrounded with mobile and embedded computing devices. Whereas these scenarios are becoming ever more realistic, programming such devices remains notoriously difficult, due to the limited resources (in terms of memory and battery power) and volatile connections these devices can sustain between each other.

This thesis will investigate code mobility to compensate for the dynamic reconfiguration of mobile networks, due to unexpected events such as device disconnections prompted by users moving about. Therefore, we propose a model for handling both device and code mobility in dynamically reconfigurable environments, such as mobile networks. In the spirit of the mobile ad-hoc networks, which are not equipped with server architectures the strong mobility mechanism performs a decentralized reconfiguration that is entirely transparent to the programmer.

Acknowledgements

I thank all the people who made the EMOOSE program possible. It was a great learning experience. I would like to thank to the people at the Ecole des Mines de Nantes for her support during all this master. I am also grateful to my professors whose teaching have been very motivating during the first part of the master in Nantes.

I wish to acknowledge the people from PROG department for supporting my project. They provided me helpful criticism, timely feedbacks and guidance on this research. I am particularly grateful for the invaluable help and dedication of Stijn and Tom. I thank them for their time, comments, suggestions and advices throughout my thesis research.

To my EMOOSE mates, Javier, Richa, Daniel and Harmin: I am grateful for their friendship. I also thank every one I got to know in Brussel; they helped me to integrate and appreciate the new environment where I had to write the thesis.

A special note of thanks to my parents, Mary and Jorge, my sister, Macarena and my son Benjamin: they have always given me unconditional love and emotional support as well as considerable encouragements. I would also like to thank Sonia for her love, her patience and encouragements during the time of hard work.

Thanks to all the people who, to some way, gave me their support for the writing of this thesis.

Contents

List of Figures

Chapter 1

Introduction

Mobile technology is changing the way computational systems serve their users. Small mobile devices provided with processing and connection capacity improve the availability of the network services. However, such devices have some restrictions that can affect their interaction over the network. These are limited resources (such as small memory and limited battery) and volatile connection to the network.

Code mobility can help the systems to reduce the impact of the dynamic reconfigurations of mobile networks, due to unexpected events such as device disconnections. In other words, device and code mobility are two phenomenons that may work complementarily for increasing the availability of the applications over the network. An application can move (or be moved) from one device to another in order to keep on serving its users. However, its mobile condition could not only be used as part of a contingency plan. This is rather a traditional vision for code mobility adopted in the context of networks with fixed participants. Code mobility can be used to create new types of services over mobile networks such as *running* applications that follow their users adapting themselves to the new contexts. Such types of applications are being developed already in the context of a new field in distributed systems known as *Ambient Intelligence* (acronym AmI) [ISTAG, 2003]. The AmI vision is that technology will become invisible, embedded in the people surroundings, whenever they need it, adaptive to users and their contexts, and enabled by simple and effortless interactions [Lindwer et al., 2003].

Next to the reason for using code mobility over mobile networks, this dissertation will focus on the properties a programming language needs to have in order to facilitate the work with both types of mobility (device and code). As final result, it will propose a mobility model for software development in mobile networks.

1.1 Motivation

As said before, some scenarios in which code mobility can be used in systems running over mobile networks, correspond to futuristic applications found in the context of AmI field. Consider the following situation:

Bob is writing a document on the PC at home. While he writes, he is listening music and chatting with some friends. Suddenly, his daughter Alice enters to the studio where Bob is and asks him for the PC to do her homeworks. Gently, Bob leaves the studio in order to take his laptop at his bedroom and continue to work there.

This story implies for Bob to reproduce somehow on his laptop in his bedroom the same work conditions he had on the PC at the studio. It is the text editor with his document, the music player with the song he was listening and his chat session. The epilog of this story usually would require for Bob to store his work and music in an external storage unit (such as a pen-drive) or somewhere on the network. After that, Bob would need to restart their applications on his laptop, copy (or download) the files and open them again.

An AmI system surrounding to Bob could lead to a different epilog:

Bob leaves the studio without doing anything on the PC, goes to his bedroom, opens the laptop (starts it if required) and after a while, the applications he was using on the PC appear on the laptop's desktop (text editor, music player and chat). He realizes that while he was changing of work place, some friends continued chatting with him. Thus, Bob answers them.

Note that what is happening in this case is that applications follow Bob, remaining available during the mobility. That is why Bob can receive messages from its friends in his chat session even during his move. Upon arrival, applications can reconfigure themselves with respect to the new resources found at the new location, in order to continue their proper operations. It is the intention of this work to demonstrate that a model that considers both, device and code mobility is good enough to develop such software applications. Of course, this scenario also requires the existence of a location-support hardware infrastructure, which is also part of the definition of the AmI vision.

1.2 Problem Statements

The problems for developing applications like the ones described above have two different sources: the properties of the mobile networks and drawbacks of previous code mobility implementations.

1.2.1 Dealing with Mobile Networks

This work consist of an analysis of the properties of mobile networks and their participants (devices), in order to identify the conditions this type of networks impose on the software development. The properties described in this work are the following:

- Devices over the mobile network are heterogenous. They can differ in their resources (screen definition, battery power, etc.).

- Devices have scarce resources and volatile connection.

- Devices provide and require services from or to other devices in the mobile network.

- Devices are autonomous and concurrent by nature.

1.2.2 Previous Code Mobility Issues

Applications available during the move can currently be implemented in different languages. However, according to the language these implementations can imply different considerations:

- If this type of application was implemented in language such as Java, it would imply to have the implementation of mobility scattered in the code. It could also imply to deal directly with topics such as concurrency (threads, monitors and synchronization) distribution (serialization, naming servers, etc.) and network organization (because partial failures).

- This type of application could also be implemented with agent-based approaches such as Telescript [White, 1996], Aglets [Lange and Oshima, 1998] and others. However, in these approaches does not provide support for discovering devices (and their services) over the network. They neither provide any mechanism for avoiding inconsistency states in the systems due to connection volatility of the devices in mobile networks.

1.3 Approach

This dissertation proposes a model for working with device and code mobility over dynamically reconfigurable environments, such as mobile networks. This model was implemented in the ambient-oriented programming (AmOP) language called AmbientTalk [Dedecker, 2005a] described in chapter 4. It is a concurrent distributed object-oriented programming language based on actors, that is specifically geared towards the use of mobile devices. As such, it provides mechanisms for dealing with their natural concurrency, volatile connections and for getting acquaintance of the services available over the network, known as *ambient* in this programming paradigm. Applications built in AmbientTalk are composed by (ambient) actors that communicate over the network in order to share their services.

The AmbientTalk mobility model presented in this work consist of the following:

1. Support for device mobility by using *Ambient References* (see section 6.2.1), an abstraction at programming language level built on top of the more primitive service discovery mechanism provided by AmbientTalk.

2. Support for actor (code) mobility in such a way, it can receive incoming messages even during the time in which it is moving from one device to another. This type of mobility is known as *Strong Mobility*. In this mobility the actor's state and its message queues are transferred, such that the communication state of any messages sent from or received by the moving actor remain intact. Strong mobility of actors is achieved by using a *move* primitive (section 6.3.1) that can be included in a higher order *move* method created by an AmbientTalk programmer.

3. Finally, this strong mobility mechanism performs a transparent (for programmers) and decentralized reconfiguration (section 6.3.3) of the communication relationships with other actors upon the arrival of the moved actor to the new location. Actors do not lose their relationships at any moment during their moves.

The mixed use of ambient references and strong mobility of actors will support the availability of applications using this model (section 6.4).

1.4 Contributions

The contributions of this thesis are the following:

- Analysis of four relevant concepts for developing applications for mobile networks: *open networks* (chapter 2), *peer-to-peer network architectures* (chapter 3), *ambient-oriented programming paradigm* (chapter 4) and *strong mobility* (chapter 5). These concepts are related to the understanding of the phenomena occurring in mobile environments (known as open networks) and the requirements imposed by this type of network (peer-to-peer architectures, ambient-oriented paradigm and strong mobility).

- Development of a mobility model that is the result of the application of the previous four concepts (chapter 6). As such, this model provides strong mobility of ambient-actors in open P2P networks.

- Development of a pattern for developing *Follow-Me* applications in AmbientTalk programming language, as the one described in the motivation (chapter 7).

1.5 Dissertation Roadmap

The chapter 2 describes a mobile network as a Open Network. It supports this match by identifying the properties of open networks in the mobile ones and their devices participants. The chapter 3 describes how the Peer-to-Peer network Architecture has been used in other types of open networks. This chapter also presents different cases in which this architecture has been applied to mobile open networks.

The chapter 4 describes the Ambient-Oriented Programming paradigm, its model for concurrency and distribution (called Ambient Actor Model) and the implementation in programming language AmbientTalk. The Ambient Actor model and AmbientTalk are compared to previous and current approaches that deal with concurrency and distribution in mobile or other types of networks. The chapter 5 defines Strong Mobility and presents some existent implementations. Some of them are direct influence of the mobility model presented in the next chapter.

The chapter 6 presents the AmbientTalk Mobility Model developed in the context of this thesis. This model is validated in chapter 7 by means of the development of a *Follow-Me* pattern for developing this type of *Follow-Me* applications mentioned above. The chapter 8 contains the conclusions and future works

Chapter 2

Open Networks in Mobile Environments

2.1 Introduction

Open networks are those that have a dynamic configuration along the time because of the volatile connection of their participants. In open networks, the participants may join and disjoin dynamically. There are well-known networks that can be considered open networks. These are the internet and the mobile networks. Traditionally, internet applications have worked under a client-server architecture. In such a case, the clients are the ones that have a volatile connection, while the servers rarely are disconnected from the network. However, a new type of architecture is getting popular in the internet: the Peer-to-Peer network architectures (the next chapter is entirely dedicated to explain this quite new phenomenon). In this case, all the participants (called *peers*) have a volatile connection. Note that in the internet, the volatility of the participants (clients, servers or peers) is mainly due to a direct decision of users to join or disjoin the network (e.g. a user opening or closing a chat or a home bank session). This is not necessarily the case of mobile networks.

Mobile networks are completely or partially composed by participants that are moving around the network. These mobile participants correspond to personal devices that move with the users. In mobile networks, a device communicate to others through a wireless connection provided by itself or by other participants specially dedicated to provide network connection. However, as it is explained in this chapter, neither the wireless connections have unlimited coverture range nor the portable devices have a huge electricity power. Thus, these networks are considered open networks because users may deliberately join and disjoin their mobile devices to the network, like in the internet case, but also because the connection of these devices depend of the coverture range of the wireless connection at the place they are being used (they can get out of range) and their power capacity. The scenario turns even more unstable if it is considered the communication between mobile devices, which is the case of one type of mobile network.

This chapter will describe the open networks in the context of mobile environments. The goal is to identify the new conditions that these networks imply for the development of applications running over them. The first part describes the hardware phenomenon implied in this case in both mobile devices and wireless network configurations. The second part discusses about the current middlewares and languages that deal with these dynamic configurations. Their drawbacks for building distributed systems for open networks are also a motivation for this work.

2.2 The Hardware Evolution

Traditional networks do not consider scenarios in which computers physically move over them. Those networks consist of a collection of fixed hosts that usually are powerful machines in terms of processor, memory, network connection and source of power. They form rigid structures offering their clients centralized services. Such systems rarely change the location of the servers. They neither add or delete servers frequently. The disconnection of a computer is considered to be abnormal if it is serving to others. It is not necessarily the case of open networks. The *join* and *disjoin* events occur much more frequently on their participants, so not all services (offered by them) are known beforehand or fixed. That is why an open network is required where participants can discover one another dynamically and where they may join or leave without causing the other participants to crash. This is the case of the mobile networks.

This section describes the hardware that enables to serve people during their movement, and the new conditions to be considered when developing systems for mobile networks.

2.2.1 Devices for Mobility

Two complementary trends can be recognized nowadays in the development of devices that support the people mobility: *mobile* and *embedded* devices.

2.2.1.1 Mobile Devices

Mobile devices are *portable* appliances with own processor, memory, source of power and wireless connection capacity. Their computational power is much lower than the power of personal computers because they are lightweight machines that can be carried and used anywhere.

Mobile devices also provide a set of perceptual resources such as screens, speakers, microphones or video cameras.

Although most of the today-devices were created to offer specific services such as cell phones, geographical positioning systems and personal digital assistants, producer companies [MIT-LCS, 2004] are arriving to provide devices with all hardware required to accomplish all these functions. Some devices that combine these functionalities are already available.

2.2.1.2 Embedded Devices

In parallel to the advances in mobile technology, another trend [Krco et al., 2005] revolves the creation of areas populated by devices embedded into physical spaces (rooms, automobiles, etc.). These embedded devices are able to interact with mobile devices in order to accomplish the services required by their users. Traditionally this type of devices have been related to sensors capable only to capture information from the environment (such as radar systems, microphones, movement detectors, etc.) [Leopold et al., 2003]. Another interpretation of this concept could also comprise it to network-connection provider devices such as wireless routers. In fact embedded devices, can play both the role of context and network provider as well as playing an active role in the delivery of services. This vision of computers into the fabric of life was called *pervasive computing* by Weiser [Weiser, 1991].

Embedded devices are provided with processing and connection capacity, and perceptual resources (as sensors). In a system running over an open network, embedded devices could perfectly execute some computations such as to determine the location of people (using mobile devices) over a network. They could also control physical entities such as lighting, door locks and heating systems.

The following section will explain how both type of devices can provide network connection to others.

2.2.1.3 Device Considerations for Software Development

Systems that consider the devices previously mentioned as participants have to consider two properties of them: *heterogeneity* and *scarce resources*.

Heterogeneity Devices can vary in the resources they provide. The resources not only come from the network, some of them are found inside of the device (like the sensors mentioned in the previous section). An application running in a device should exploit the particular features of the devices to provide the appropriate set of interactions. Note that the features of a device can determine not only the user interface aspects. Restrictions in the processing power or operating memory make heavy applications unsuitable in devices. According to [Islam and Fayad, 2003] most of systems have wrongly used the model-view-controller pattern to deal with device heterogeneity (in which only the view varies). It is not enough since data or model used devices can be different too. As is explained in the next section, restrictions in the network connection (phenomenon known as *connection volatility* [Dedecker, 2005a]) imply to modify the concurrency and distribution model.

Scarce resources Devices for mobility have advanced rapidly in terms of available resources, however they still remain limited when compared to PCs and laptops. The biggest improvement to the hardware of these devices are in the processing, memory and connection capacities. Unfortunately the sources of power (battery technology) has not followed the same trajectory. A system developed for such devices should provide some alternatives to reduce the impact of the unexpected changes in the resources of the devices (loss of connection, depleted battery, etc.). It could provide mechanisms for storing the application in memory (the data and/or the state), or moving a part of it to another device in order to ensure the availability of the service provided by the program.

2.2.2 Wireless Networks

The second fundamental component of a open network is the physical network that supports the mobility of the people and their devices. According to its composition a network can be defined either as a *Nomadic* or an *Ad-hoc* network [Mascolo et al., 2002]. This section also describes two phenomenons related to these networks. The first one explains the mobility of a whole network (with all its devices) around other networks (*network mobility*). The second one refers to the fact that the environment where devices move around can be composed by a set of *overlay networks*. As it is explained in chapter 7, these two cases imply particular requirements to the systems developed for mobile networks. As in the previous section, the final part will describe the implications for the software development of the different types of networks, as well as the mentioned phenomenons.

2.2.2.1 Nomadic Network

Nomadic systems are composed of a set of mobile devices and an infrastructure with fixed nodes. These fixed nodes are responsible to provide wireless network connection to the mobile devices. The fixed devices can be servers or desktop machines, or can be embedded devices in the sense explained in the previous section. Fixed devices can also be wired *access points* like wireless routers. Depending of the size of the network, it can have more than one access point providing network connection. It ensures mobile devices to have connection along the boundaries of the space covered by these providers. Note that the same low-level mechanism used for mobile devices to switch transparently of access points inside a network (known as *handoff* or *roaming* [Lin and Chlamtac, 2001]), could be used to switch from a network to other, or to choose one in case of having overlay networks. The only visible boundaries for the people are those related to commercial or political reasons [Cardelli, 1999].

Figure 2.1 shows a generic scheme of a nomadic network. Some cases of nomadic networks are the cellular networks and wireless LANs. In addition, it will be presented a emerging case a of wireless sensor networks that matches properly

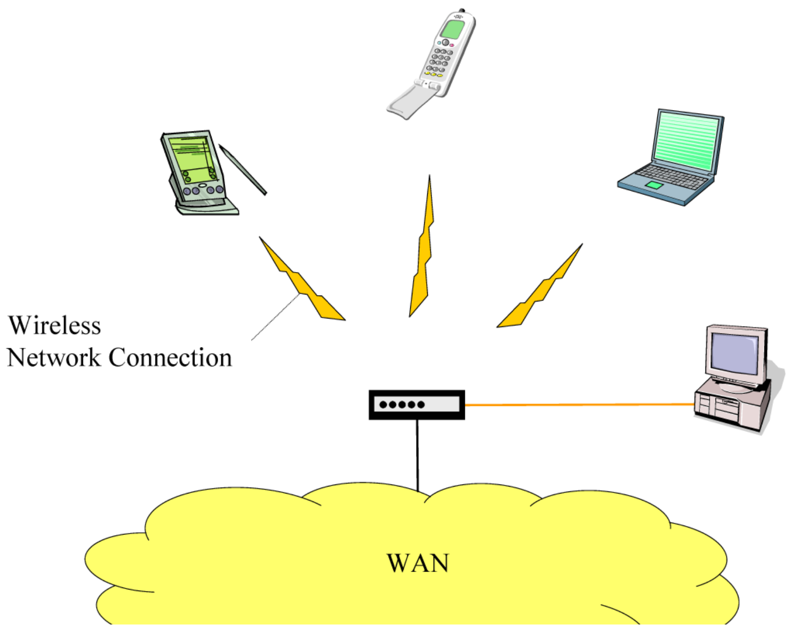


Figure 2.1: Nomadic Network

with the definition of nomadic networks.

Wireless LAN A wireless LAN or WLAN is a network infrastructure with fixed and wired access points (also called base stations or gateways). In a WLAN the last link with the mobile device is wireless, giving it network connection in the surrounding space. Mobile devices within this network connect to the nearest base station that is within its communication radius [Wang, 2003].

The first versions of WLANs bridged these networks to a wired Ethernet networks. Actually, WLAN are also supported by stand-alone base stations such as broadband/ADSL connection boxes.

Cellular Network Cellular networks form a nomadic network between the cellular phones and their base stations. A mobile phone is a node with a *mobile IP* that provides it with two IP address: a fixed home address and a care-of address that changes at each new access point attachment. This type of network is often found questionable because the high latency and disruption produced by frequent handoffs (changes of base station) of mobile phones.

Wireless Sensor Network A wireless sensor network consists of a number of *smart sensors* that are embedded devices with sensing capabilities, able to perform data processing tasks and wirelessly communicate with other devices [Berger et al., 2003].

As sensors, these device are equipped with perceptual resources such as microphones and antennas. There are several experiments in which wireless sensor networks are used to interact either with mobile devices or people directly. These networks can be used for locating mobile devices over the network. It will be briefly explained in the chapter of application and validation of this work (chapter 7). There are other experiments in which mobile devices have been provided with sensing capabilities and therefore people can scatter those devices across an area of interest [Priyantha et al., 2000].

2.2.2.2 Ad-hoc Networks

Ad-hoc networks or mobile ad-hoc networks (called MANET) were initially developed by DARPA project in the early 1970's. MANETs are mobile wireless communication networks without infrastructure support. Each one of the participants can act as router to other nodes in the network and cooperates with other nodes within its range to discover and communicate each other. Both discovery and communication are done without a central server and predefined central access points.

A MANET has a dynamic configuration along the time because the constant arrivals and departures of mobile devices. It requires the MANET to have network protocols that enable it to organize itself after each change in the network. Device mobility can also produce variations in the network connection quality, if these devices are supporting such connection. Nevertheless, there are as many other potential network connection routes as devices with connection capacity are joined on the MANET. This information is also managed by the network routing protocols.

Figure 2.2 depicts an ad-hoc network. A particular case of it can be found in a *personal area network*. This concept invented at IBM [Zimmerman, 1996] is described at following.

Personal Area Network A Personal Area Network (acronym PAN) is defined as the network composed by all devices that a person carries while he moves around. These devices (laptops, cellular phones, PDAs, digital cameras, etc.) can conveniently be connected to cooperate each other during the mobility. This cooperations can be simple ones such as moving information from a digital camera to a laptop, or can be more sophisticated ones such as devices providing network connection to the others device of the PAN (which is actually the behavior of a MANET).

2.2.2.3 Network Mobility

A network and all its devices can move as single unit over one or many bigger networks. There are several cases in which (nomadic and ad-hoc) network mobility can be seen. These cases even can happen at the same time:

- A PAN moving with its user.

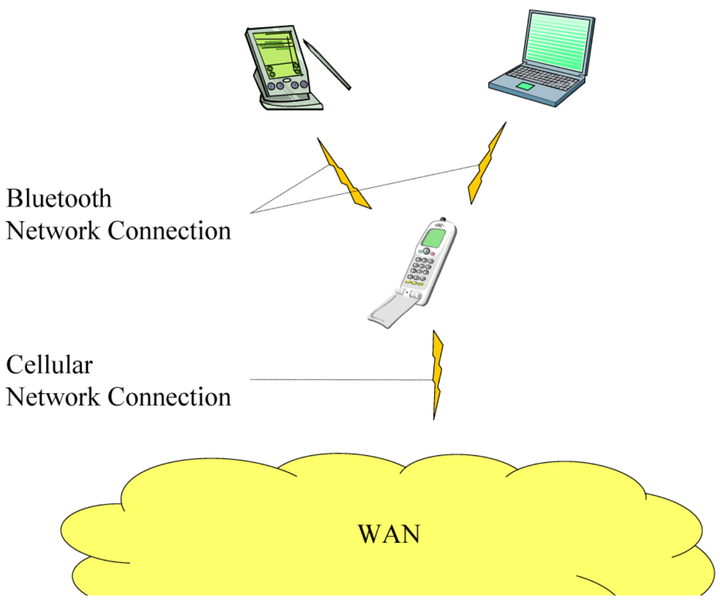


Figure 2.2: Ad-hoc Network (MANET)

- Wireless sensor networks deployed in vehicles providing driving facilities.
- Access network deployed in public transportation (buses, trains, aircrafts). Note that a user of this public transport could be using a PAN.

Actually there are several projects that are dealing with this in order to make transparent for the users all the handoffs implied in the mobility [Lin and Chlamtac, 2001]. The network also has to be managed at the application level because the available resources as well as the permissions of the networks can change from one to another.

2.2.2.4 Overlay Networks

People can move over places in which one or more of their devices can find one or more overlay networks. These networks can be both nomadic and ac-hoc networks. It could be an advantage for users if the system managing the connection of the device provides a mechanism to permanently choose the best connection.

It seems to be a subject for network developers. However, as in the case of the network mobility, it can be also a matter of the application level. The following section will explain why.

2.2.2.5 Network Considerations for Software Development

At least four new properties need to be considered for the software development after having described the nomadic and ad-hoc networks. These are based on the description of hardware phenomenon found in [Dedecker, 2005a]. The considerations are *connection volatility*, *ambient resources*, *device autonomy* and *natural concurrency*.

Connection volatility It is related to the fact that a connection between two devices can never be considered stable. Wireless networks are unpredictable and can induce long delays on the networks traffic. In addition, the mobility of users can imply that their devices get out of range. In many cases this disconnection is only temporal. It would be desirable that applications that became disconnected can somehow *resume* their communications once they are reconnected.

Another desirable property of the systems is the possibility for users to move information or running applications from remote devices to their current devices. Thus, they actively help to ensure the availability of the services they require.

Ambient Resources The resources an application needs to perform its tasks can be in the devices containing it or in other devices on the network. It implies that resources can become dynamically (un)available according to the connection-state of the devices that contain them. [Dedecker, 2005a] calls the available remote resources *ambient*, term understood as the place where the computations happen. This concept is explained in more details in chapter 4.

It was previously explained that devices can move around overlay networks. Normally it would be desirable that they have always the *best available connection* in terms of quality (speed, reliability, etc.). However, the best connection will depends also of the devices (and their resources) that can be reached using this connection. In such a case, it would be useful to allow users to choose personally the most convenient connection according to their purposes.

Autonomy *Every device acts as an autonomous computing unit* [Dedecker, 2005a]. The dynamic configuration of the wireless networks (especially the ad-hoc networks) makes rigid client-server approaches unsuitable. In a system composed entirely of devices that are constantly joining and leaving the network, it is impractical to have some devices with higher responsibilities (since it can disappear at any time as any other device). Instead, the responsibilities in a network can perfectly be shared by its participants.

Natural Concurrency Another property of mobile devices (additional to those described in section 2.2.1.3) is their natural concurrency. Every system that implies the participation of several devices, is considered a concurrent system, since each device has its own processor. It implies for the system to implement communication systems that exploit this natural concurrency maximally.

2.3 Systems for Open Networks

The following sections will introduce some current languages and middlewares that can be used to build systems for open networks.

2.3.1 Languages

The inclusion of mobile devices to the networks separates the programming languages that do not support this mobility from those that support it (or could do it according to their characteristics). The former languages have been identified by [Dedecker, 2005a] as languages for local area networks. The latter ones will be consequently the languages for open networks. Most of the following languages will be explained in more detail in chapters 4 and 5.

2.3.1.1 Languages for Local Area Networks

These languages were conceived for reliable networks, with centralized services offered by privileged (in term of network and computing capacity) servers. The disconnection of these servers is considered harmful for the system.

On another hand, some of them take advantage of the network reliability and offer synchronous communication primitives. As it will be explained in chapter 4, synchronous communications is harmful for the autonomy of mobile devices. Some languages using synchronous communications are Java [SUN Microsystems, 2005], Emerald [Jul et al., 1988] and Obliq [Cardelli, 1995].

2.3.1.2 Languages for Open Networks

These languages were not conceived to deal with mobile devices. Some of them still do not do it. However, they have more adequate properties for supporting the dynamicity found in an open network. First, most of these language are based on the actor model concurrency model. As it is explained in chapter 4, it uses asynchronous communications which preserves the autonomy and natural concurrency of the devices. However, as remarked in [Dedecker, 2005a] these languages either do not consider mechanisms to discover ambient resources or do not deal with the connection volatility. Some of these languages are E [Miller, 2000], SALSA [Varela and Agha, 2001], ChitChat [De Meuter, 2004].

2.3.2 Middleware

Several recent research works have focused on the extension or creation of middleware for supporting open networks such as nomadic and ad-hoc networks. However, it has not been straight forward task. Traditional middleware has characteristics that make it unsuitable in this new context:

- Mobile devices require light computational load. Existing middleware usually requires a heavyweight platform that cannot be deployed on these devices.
- As will be largely explained in chapter 4 the connection volatility of devices require asynchronous communication. Traditional middleware does not consider this type of communication.
- Systems in open networks run in an extremely dynamic context which requires devices to be aware to their environment. Existing middleware were built thinking in terms of fixed distributed systems. There is no support for this kind of ambient-awareness.

According to [Mascolo et al., 2002] the recent works on middleware can be split into different research areas such as *RPC-based*, *publish-subscribe*, *tuple space-based* and *data sharing-oriented* middleware.

RPC-based middleware Attempts to make CORBA suitable for nomadic networks focus their work on making ORBs suitable in mobile device and the IIOP protocol resilient to failures in the communication. Other RPC-based approaches support queuing of RPCs or enable rebinding of resources. These approaches have gotten good results only for short-time connections.

Publish-subscribe middleware This middleware included the event-based interaction. Consumers subscribe to events they are interested in and they are notified when they are published. This mechanism allows to express context-aware communications. However, the communication is made via *callbacks* which hamper the understanding of the program.

Tuple space based middleware Tuple spaces [Gelernter, 1985] have proven suitable for mobile computing because of the dynamic context nature of mobile systems. They can be used for coordinating mobile units across a mobile computing environment. However, this paradigm does not integrate well with the object-oriented paradigm.

Data sharing-oriented middleware This middleware tries to maximize the autonomy of the devices by introducing weak replica management facilities [Dedecker, 2005a]. The drawback is that weak replicas are not always synchronized because of the connection volatility of the devices. The conflicts produced must be resolved depending on each application specific needs.

2.4 Conclusion

This chapter has described the open networks and has identified the implicants for the software development. These implications are:

1. A system running over an open network should exploit the heterogeneity of the devices.
2. A system should be aware of the condition of the resources of the devices, and provide some mechanism to ensure the availability (if possible) of the services of the devices.
3. A system should be connection-aware, able to deal with the connection volatility of devices.
4. A system has to be ambient-aware, able to discover the resources present in the environment.
5. A system must respect the autonomy of every device. Note that in a open network every device can be defined in terms of the services it offer and those it requires. As such, devices may perfectly be used as client (requiring services) and server (providing services) at the same time.
6. Devices are concurrent by nature (they have their own processors). They have to work together in such a way that their natural concurrency is not hampered but rather exploited.

The open networks in mobile environments are the scenarios targeted in this work. The three following concepts (explained in the three following chapters) are included to support the considerations of open networks above mentioned.

The next chapter describes the *Peer-to-Peer network architectures*, a phenomenon that has been evolving in parallel to open networks, in another context. This chapter will argue why this type of architecture is the most adequate for the development of distributed systems in open networks.

Chapter 3

Open Peer-to-Peer Network Architectures

3.1 Introduction

The main purpose of this chapter is to describe the peer-to-peer network architecture and its close relationship with the open networks along its evolution, initially with the internet and currently also with mobile open networks.

The Peer-to-Peer (acronym P2P) concept means to share responsibilities of a work between the participants in a network. Unlike in client-server architectures described in section 3.2.2, everyone can serve and be served in P2P schemes, even at the same time.

P2P computing is not new. [Minar et al., 2001] state the internet was initially conceived to have P2P relationships between the users, arguing that still working applications such as Usenet and DNS (explained in section 3.2.1) were conceived under this architecture. Early versions of these applications already had some of the properties observed in recent P2P applications, like decentralization and scalability. As it is briefly explained in section 3.2.2, the boom commercial of internet in the nineties produced the changed of its shape, imposing the client-server architecture.

Nowadays, when file sharing applications have brought back P2P concepts, these systems show strengths never seen before in client-server structures. To attributes already mentioned of early P2P systems (decentralization and scalability), other new properties are added. Current P2P applications are also recognized as self-organized and fault-tolerant. As it is explained in sections 3.3 and 3.4, these properties are the solution to the condition of open network of the internet. The previous chapter identified an open network as a network with a dynamic configuration along the time because of the volatile connection of their participants. This is exactly the problem that file sharing applications solve by using a P2P architecture.

The final part of this chapter will discuss the result of some cases in which this architecture has been used for mobile open networks.

3.2 P2P Systems and the Internet

As mentioned above, a P2P setup is not limited to the scope of internet; however this network has been the test bed along its history for applications that have used the P2P scheme. The following subsections identify some stages in the evolution of the internet and its relationship with the P2P architecture, in order to clarify the context of the P2P paradigm.

3.2.1 Early P2P Systems

In the early days of the internet several P2P applications were developed, which typically operate of the level of ISP's server infrastructure[1]. The two most popular cases are Usenet and DNS [Minar et al., 2001].

3.2.1.1 Usenet

Usenet [Minar et al., 2001] is considered the first file-sharing protocol (still in use since 1979). It was originally based on a mechanism (Unix-to-Unix-copy protocol) by which a machine could automatically dial another, exchange files and disconnect when the exchange ended. Its most popular use has been as a news system. At present, Usenet works on a TCP/IP protocol known as the Network News Transport Protocol (NNTP), which allows news servers on the Usenet network to discover newsgroups efficiently and exchange new messages in each group.

In Usenet there is no central authority that controls the news systems. For instance, each new newsgroup addition to the main topic hierarchy is proposed and discussed in the Usenet group *news.admin* in a *rigorous democratic process* [Minar et al., 2001].

With respect to NNTP, this protocol contains meta-data such as the *path header* in the news messages to trace their transmission from one news server to another. It avoids a flood of repeated messages during NNTP transmissions: A server will not try to send a message to another if it is already contained in the path header of such a message. Optimizations similar to the path header were not used by more recent P2P systems like Gnutella (see section 3.3.4). Consequently, a Gnutella node could receive the same request repeatedly.

Usenet is a successful case of an open and decentralized system. But this decentralized nature became a problem because of the commercial internet explosion. Spam made Usenet a extremely noisy communication channel. Nevertheless, Usenet is a good design lesson about P2P systems for its decentralized control, methods of avoiding a network flood and other characteristics, even with its current problems.

[1]Internet Service Provider, a company that provides access to the Internet.

3.2.1.2 Domain Name Systems

A Domain Name System [Minar et al., 2001] (acronym DNS) stores the information about host and domain names. This system blends P2P networking with a hierarchical model of information ownership. Its best known property is scalability, from few thousand hosts in 1983, to hundreds of millions of them currently on the internet.

DNS was established as another solution to the file-sharing problem. The way to map to a human-friendly name to an IP address was through a single text file (host.txt) which was copied around the internet periodically. Since this file became unmanageable because of the explosive growing number of hosts, it was necessary to develop this system (DNS) to distribute the data sharing across the internet.

The Namespace of DNS names is built hierarchically. This hierarchy yields a simple natural way to delegate responsibility for serving part of the DNS database. Each domain has a name server (known as *authority* [Minar et al., 2001]) of records for hosts in that domain. Looking for the address of a given name, the server can query to its nearest name server, or delegate the query to the authority for that namespace. Successively the query can be delegated to a higher authority up to the root name servers for the internet as a whole. When the answer is achieved, it is propagated back down to the requestor. The result is cached along the way to the name servers in order to make the next fetch more efficient. Name servers play both roles to do this work, servers and clients. This is one of the design decisions that have helped to make the network more scalable. The second decision is the natural method of propagating data request across the network. Although a DNS server can query any other; there is a standard path up the chain of authority. The load is naturally distributed across the DNS network. Any individual name server needs to serve only to its clients and the namespace it individually manages.

3.2.1.3 Evaluation of Early P2P Applications

The properties recognized in these two early P2P applications are the following:

Decentralization There is no a central control in both applications. Usenet has a democratic administration process and in the DNS case each name server is server and client at the same time. Note a client-server scheme could be understood as a particularization of a P2P architecture: a client is a peer with exclusive responsibilities of a client, and the opposed case happens with a server peer.

Scalability It is related to the capacity of a system to increase its number of participants without any traumatic reconfiguration process. It is achieved in the DNS by its chain of authority and its peers with client and servers responsibilities.

3.2.2 Client-Server Network Architecture

The explosion of the internet in 1994 radically changed its shape, turning it from a quiet geek utopia into a bustling mass medium [Minar et al., 2001]. The millions of people interested in sending emails, viewing web pages, downloading files and buying products had a far-reaching impact in the way network architecture evolved. *These changes affected directly the original conception of internet as a P2P network* [Minar et al., 2001].

The network model of software development changed significantly, not just for the bandwidth consumption, but also for the methods of addressing and communicating in the network: Modem connection protocols (as PPP[2] and SLIM[3]) became more common; applications were created according to users with slow-speed connection; and companies started to manage their own networks shielded through firewalls and NATs[4]. These changes were made according to the usage patterns common at that time, mainly thinking in downloading files, not uploading or publishing information.

The web browser was conceived under a client-server network architecture, in which the client connects to a well-known server, downloads some data and disconnects. In this operation the web client needs neither to have a permanent or well-known IP address, nor a continuous connection to internet, nor to accommodate multiple users. It just needs to know to who and how to ask a question and listen for a response.

It was explained in the previous chapter that internet is an open network. It has a dynamic configuration over the time because of the volatile connection of the (client) computers used by people. Usually it is not the case of the servers whose have a stable participation on the internet. Note that client-server systems depends mainly of the stable connection of the servers.

3.2.3 Open P2P Network Architecture

P2P architectures came back to the internet with the popular file sharing applications at the end of 1990s. These applications were developed to enable the user's PCs to interact directly to each other without the intervention of a server. Figure 3.1 depicts both the client-server and the P2P network architectures.

File sharing applications had to deal with different conditions to ones found in the context of the early applications. Systems like DNS rarely has to deal with

[2]Point-to-Point Protocol commonly used to establish a direct connection between two nodes. It has been used to connect computers trough phone line. ISPs use this protocol to offer dial-up internet access. [Wikipedia, 2004]

[3]The Serial Line Internet Protocol (SLIM) was an encapsulation of the internet protocol designed to work over serial ports and modem connections. [Wikipedia, 2004]

[4]Network Address Translation, also known as Network masquerading or IP-masquerading, is a technique which IP address of network packets are rewritten as the packet pass through a router or firewall. It is used to enable multiple host on a private network to access the internet using a single public IP address. [Wikipedia, 2004]

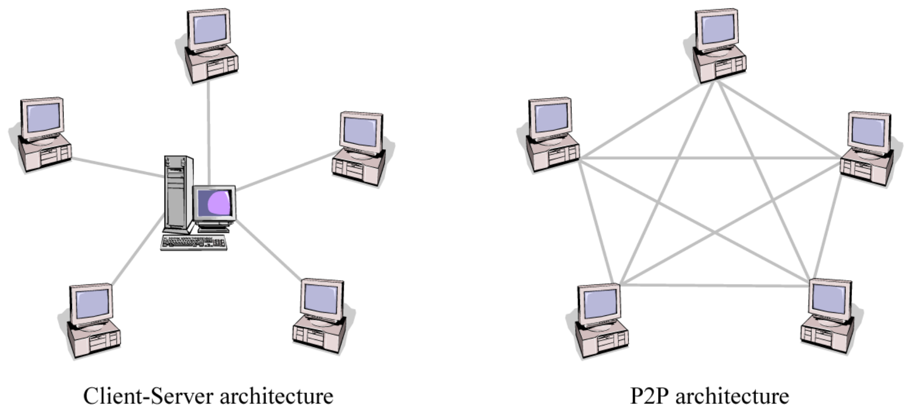


Figure 3.1: Network Architecture

the disconnection of their participants (name servers). It is not the same scenario for file sharing applications. Their participants (peers) are computers managed by users that may voluntarily leave or join the network at any moment. Thus, These applications have had to develop mechanism for dynamically discovering peers, organizing themselves after the connection and disconnection of a peer and routing messages between them. All these mechanism will be detailed in the following two sections.

3.3 Open P2P Applications

This section describe the different types of P2P network architectures that have been developed in the past few years. Nowadays it is possible to find different types of P2P applications such as file sharing systems [J. Liang and Ross, 2004, Cohen, 2003], some instant messaging systems [Cerulean Studios, LLC, 2005], distributed computing [PlanetLab Consortium,] and others. This chapter will focus the attention in file sharing systems uniquely because such applications are illustrative for all the types of open P2P network architectures (even for mobile open P2P networks, as it is explained in section 3.6). A file sharing system make files available to users in a network. In such systems each peer can both download and upload files the same time.

[Eberspcher et al., 2004] identifies four types of open P2P network architecture: centralized, hybrid, pure and DHT-based P2P network architectures See figure 3.2 (DHT-based P2P architectures are depicted in the following section).

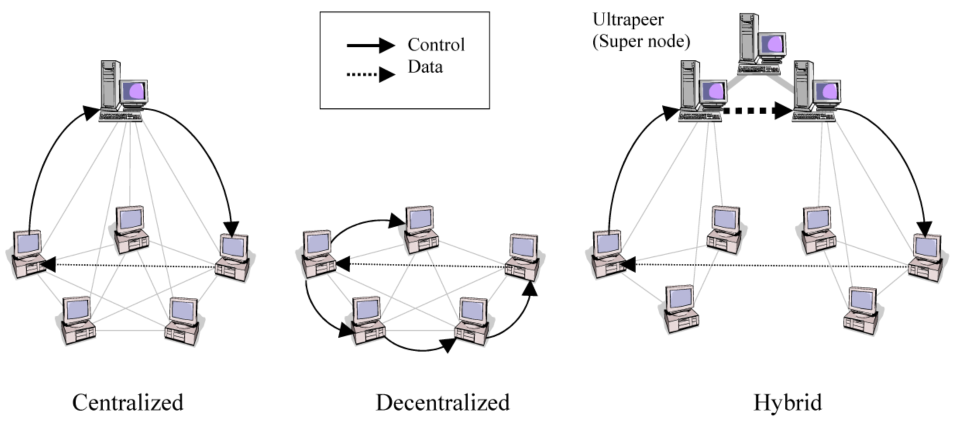


Figure 3.2: P2P Network Architectures

3.3.1 Centralized P2P Applications

These systems are also known as *mediated systems* [Susan Crosse and Smith, 2003]. They use a client-server setup for its control operations (figure 3.2). Each peer must logon to a central server. In most of the cases servers manage a database with all connected users and their shared files. Searches are sent to the server and if it finds the required file, searcher peer can download the file directly from the peer that has this file.

Napster 1.0 Napster 1.0 was an online music service created by Shawn Fanning in 1999 (Northeastern University, USA). It was a mediated P2P network consisting of a centralized server for performing search functionality [Susan Crosse and Smith, 2003]. It used a client-server protocol over point-to-point TCP for performing the searches. Napster does not provide a complete solution for bypassing firewalls, and it's capable of traversing only a single firewall. This way, each peer acts as a simple router, capable of sending content to a firewalled peer when a request is made via HTTP. As previously mentioned, a centralized system has a single logical point of failure. Napster can load balance among servers using DNS rotation. However, this was considerated a potential congestion point.

Evaluation of Centralized P2P Systems Centralizing logging and searching activities in these systems processes an effective access control. However, they inherit the traditional problems found in client-server architectures, such as having a single point of failure, performance bottlenecks and scalability.

3.3.2 Decentralized P2P Applications

These systems are purely P2P. They don't use a central server at all. Each peer has the same responsibility: There are no group leaders. Queries for files are broadcasted through the network. Pure P2P networks have become unpopular because they generate a lot of overhead traffic to keep them up and running. Nowadays, Freenet still uses this model because it offers an unprecedented anonymity.

Case: Gnutella Gnutella is a distributed software project to share files in a pure P2P network. It was invented at Nullsoft by Justin Frankel and Tome Pepper (March 2000). Current versions of this protocol run over TCP/IP. In Gnutella network, searches (queries) are propagated from a peer to all their known peer neighbors. The response is routed back using the same path. When a resource is found and selected for downloading, a direct point-to-point connection is made between the client and the host of the resource. The file downloaded directly using HTTP.

Evaluation of Decentralized P2P Systems In a Pure P2P architecture all features of the system rely on the peers. Pure P2P systems have good performance in small networks. In contrast to the centralized systems (with centralized control access), security is often a very important issue in these systems.

3.3.3 Hybrid P2P Applications

Hybrid Architectures tend to get the best of the both alternatives mentioned before, through the introduction of *ultrapeer* concept [Susan Crosse and Smith, 2003] (also known as *super nodes* [J. Liang and Ross, 2004]). An ultrapeer accomplishes the role of a server like in centralized systems, but only for a limited number of peers. In this scheme there are a set of ultrapeers themselves connected through a pure P2P network. This is the reason why hybrid systems have two layers in the control plane (figure 3.2): one in with peers connected to one ultrapeer (in a client-server fashion) and another with ultrapeers connected in a decentralized P2P network. Both build an overlay network over existing IP network.

Case: KaZaA Kazaa media desktop is a file sharing application that uses the FastTrack protocol. Both were created at company Consumer Empowerment by the Niklas Zennstrm and Janus Friis (March 2001). FastTrack is based on the Gnutella protocol; it extends this protocol with the addition of Super Nodes for improving scalability. Super Nodes (acronym SNs) are more powerful than Ordinary Nodes (acronym ONs). As said before, ONs are assigned to a SN. Every node can become SN automatically if it has enough bandwidth and processing power [Parashar, 2004]. Each SN maintains an overlay network with other SNs (long-lived TCP connections between them).

Two-tier hierarchical networks work more efficiently in large-scale systems. Since there is often more heterogeneity in peers in such systems. Peers can differ in up-times, bandwidth connectivity and CPU power accessibility (behind NATs or firewalls). To exploit the heterogeneity, the organization must consider a hierarchy where nodes more powerful (in terms of the properties mentioned) are automatically allocated in the higher tier.

Case: Bit Torrent BitTorrent is a P2P file distribution application created by Bram Cohen under MIT License (February 2002). This tool allows people to download the same file without slowing down everyone else's download [Wikipedia, 2004]. It is possible by swapping portions of the file between the downloaders. Peers frequently connect for small portions of time.

BitTorrent uses tit-for-tat method to seek Pareto efficiency [Cohen, 2003]. Tit-for-tat is a strategy in game theory for the iterated prisoner's dilemma (to maximize his own advantage, without concern for the well-being of the other player) It consists of responding in kind to a previous opponent's action [Wikipedia, 2004]. An allocation is Pareto efficient if there is no other allocation in which some other individual is better off and no individual is worse off. [Osborne, 1997]. Downloaders are encouraged because every client uploading to others gets faster downloads. This way the system achieves a higher level of robustness and resource utilization than current cooperative techniques. In traditional P2P systems there are problems of fairness between peers in the network. BitTorrent proposes that each peer's download rate has to be proportional to their upload rate.

Evaluation of Hybrid P2P Systems As in the case of the decentralized systems, Hybrid P2P applications are highly scalable and fault-tolerant. In addition, experiments with hybrid solutions [Parashar, 2004, Cohen, 2003] have concluded that these are much more efficient than both centralized and decentralized systems. However, security problems remain.

3.3.4 DHT-based P2P Applications

Currently, it is possible to find some highly structured P2P network architectures based on Distributed Hash Tables (DHT). In DHT-based architectures, every peer (called *node*) is assigned a unique key by a hash algorithm. The keys, along with the network address of the peer are evenly distributed among all connected peers. Each peer maintains a routing table and queries are only directed to those peers in the routing table. DHT-based algorithms are aimed to be fast and accurate in the peer lookup over a network [Ding and Bhargava, 2003].

The next section will describe briefly some of the best-known DHT-based algorithms.

3.4 P2P Algorithms

This section present three of the best-known DHT-based algorithms namely Chord, Pastry and CAN. All these algorithms will be discussed in terms of their naming, routing, node addition and node failure mechanisms.

3.4.1 Chord

Chord [Ion Stoica and Balakrishnan, 2001] was developed at Massachusetts Institute of Technology (MIT). Chord is an algorithm that uses a ring topology. The basic idea of Chord describes how nodes join the ring, how data is stored and how the ring recovers from node failures. Interfacing with applications, Chord provides only one function: given a key, it maps the key onto a node. That node would typically be responsible for storing the data associated with that key, or it could store information about where that data can be found.

3.4.1.1 Naming and Routing

The hash function begins by assigning to nodes and keys an $m-$bit identifier, organizing the nodes in the ring based on the node identifiers. Both keys and node identifiers map onto the same ring and the first note that follows a certain key in the ring is responsible for that key. Each node keeps track of its successor and predecessor on the ring.

Figure 3.3 shows a Chord network with three nodes identified for 0, 1 and 3. The key identifiers are 1, 2 and 6. The successor of key 1 among the nodes in the network is node 1, so key is stored at this node. The successor of key 2 is 3 because is the first node found moving clockwise from 2 on the identifier. Finally, successor of key 6 is the node 0.

In addition to ring strategy, Chord provide the nodes with data structures called finger tables. In these tables only the successor is needed for a correct search-result (however, the predecessor makes it possible to efficiently join and leave the network, see following sections). Each node keeps a number of pointers (periodically checked and updated if possible) to nodes at an exponentially increasing distance away.

3.4.1.2 Join

A node joins a network by asking an arbitrary node to look up its successor, updates predecessor and successor pointers and finger-tables entries, and transfers the

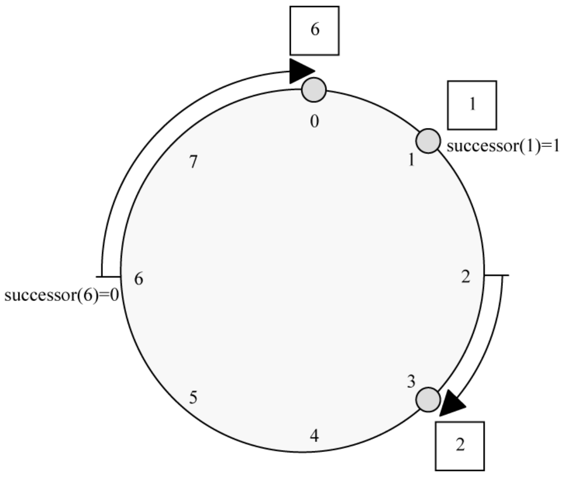


Figure 3.3: Hash Function in Chord

relevant keys from its successor. The node (that receives the petition from the new node) proceeds to send notifications. It notifies to the nodes that supposedly have itself in their finger table. This is done by notifying all nodes that are predecessors of keys which are 2^i steps before the node that just joined. This may occasionally miss, but that is only relevant for performance and will be corrected on the next periodic update.

3.4.1.3 Failure

Failures are detected periodically by a stabilization protocol. In the event of a node failure the ring can be traversed via the successor pointers only. In order to keep the ring intact, each node keeps track of a number of its immediate successors. If a node is detected as failed it will be removed from the ring. The extra successor pointers are used for this. The fix-fingers routine is executed periodically to make sure that the finger tables are correct and up to date. If a node in the path to the node that holds a specific key is broken, the lookup will fail. This can be handled by retrying after a short while, since the stabilize protocol will repair the ring. A different approach is to ask the previous entry in the finger table. This should give a new and hopefully valid path, even though performance would decrease. It is also possible to keep track of the predecessors to the entries in the finger table.

3.4.2 Pastry

Pastry [Rowson and Druschel, 2001] is a project initiated at Microsoft Research focusing on P2P anonymous storage. It is a scalable, distributed object location and routing substrate for wide-area P2P applications. It follows two goals with its DHT-based routing approach. Firstly, just like in Chord, Pastry provides methods to be able to route to any shared node available and identifiable (with a key). Secondly, it takes into account network locality. Thus Pastry minimizes the distance the message travels, according to a scalar metric, like the geographical distance, the distance within the IP network in terms of hops or delay.

3.4.2.1 Naming and Routing

Each node has a unique 128-bit node identifier. The node identifiers are uniformly distributed by basing them on secure hash of the node's public keys or IP addresses. The node with a node identifier that is closest to a given key is responsible for that key, but the key is also replicated to a certain number of the nodes in the leaf set. When a request for a specific key arrives at a node, this node forwards the request to a node that shares one digit more in the prefix with the given key. If there is no such node in the routing table, the message is forwarded to a node that shares the same prefix but whose node identifier is numerically closer to the key. This way the request is forwarded closer and closer to the responsible node. Figure 3.4 shows a path example of a message. Since Pastry routes requests to a node that supposedly is close in the network layer, it is likely that requests first arrive at a replica which is closer to the requesting node than the responsible node.

3.4.2.2 Joining

The efficient way how Pastry maintains dynamically the node state is key to its design (node state comprises *the routing table, leaf and neighborhood sets, node recovery in case of nodes failures, and new nodes arrivals.* [Rowson and Druschel, 2001]). When a new node X arrives, it receives a new node identifier. It needs to initialize its state tables, and then inform other nodes of its presence. Node X knows initially about a nearby node A (according to the proximity metric). So, X asks A to route a special join message with the key equal to X. Pastry routes the message to the existing node Z whose nodeID is numerically closest to X. In response, X obtains all state tables of encountered nodes in the path from A to Z, information used by X to initialize its state.

3.4.2.3 Failure

Respect to nodes departure, a Pastry node is considered failed if its neighbors can no longer communicate with it. Then, all members of the failed node's leaf set are notified and they update their leaf sets. This action is trivial because the leaf sets of nodes have adjacent node identifiers overlap. A recovering node contacts the

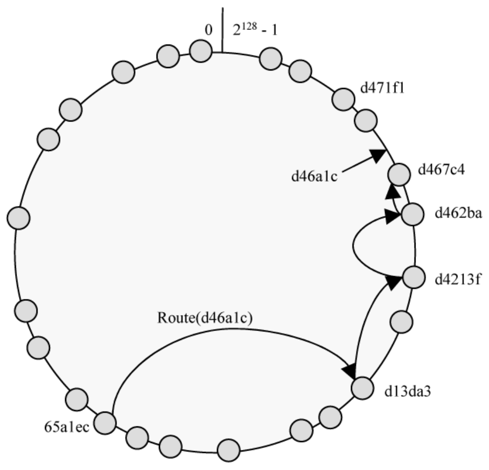


Figure 3.4: Routing a message in Pastry

nodes in its known leaf set, obtains their current leaf sets, updates its own leaf set and then notifies the members of its new leaf of its presence. Routing table entries that refer to failed nodes are repaired lazily.

3.4.3 CAN

Content-Addressable Networks [Ratnasamy et al., 2001] (acronym CAN) was created at AT&T Center for Internet Research at ICSI. The goal of this algorithm is to improve the performance of large-scale distributed systems (such as Internet systems). The main feature of CAN is the mapping of a key k onto a point P in a $d-$dimensional cartesian coordinate space. The coordinate space is partitioned among all nodes in the CAN so that each node is responsible for a zone. Figure 3.5 shows a $2-$dimensional $2-$bit coordinate space partitioned between 5 CAN nodes.

3.4.3.1 Naming and Routing

CAN works by following the straight line path through the Cartesian space from source to destination coordinates. Nodes maintain coordinate routing tables that hold the IP address and virtual coordinate zone of each of their neighbors in the coordinate space. In a $d-$dimensional coordinate space, two nodes are neighbors if their coordinate spans overlap along $d-1$ dimensions and differ along one dimension. The entire CAN space is divided amongst the nodes currently in the system

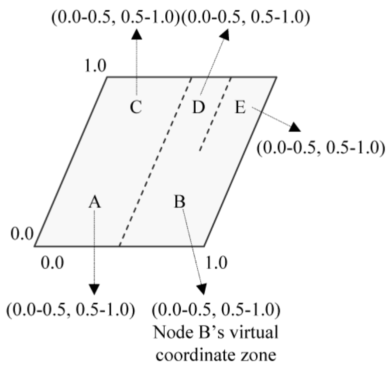


Figure 3.5: $2-$dimensional cartesian coordinate space

[Ratnasamy et al., 2001]

3.4.3.2 Joining

When a new node joins the CAN, it randomly chooses a point P in the coordinate space and sends a `join` request for P to any known node in the network. The `join` request is routed to that node in which zone P is located. The joined node is allocated by splitting this existing node zone in half, retaining half and handing the other half to new node. This process happens as following:

- First the new node must find a node already in the CAN.
- Next, using the CAN routing mechanisms, it must find a node whose zone will be split.
- Finally, the neighbors of the split zone must be notified so that routing can include the new node.

3.4.3.3 Failure

CAN ensures in case of nodes departures, that zones occupied for them are taken by remaining nodes. To do this, the node has to hand over its zone and the associated database (key, value) to one of its neighbors as following:

- To merge its zone with his neighbor's one.

- If the first alternative does not produce a valid zone, handing the zone to the neighbor whose current zone is smallest, and the node will the temporarily handle both zones.

In case of node or network failures (one or more nodes become unreachable), CAN handles this through an immediate takeover algorithm that ensure one of the failed node's neighbors takes over the zone. In this situation the pairs (key, value) held by the departing node would be lost until the estate is refreshed by the holders of the data.

3.4.4 Evaluation of P2P Algorithms

P2P network architectures based on DHTs have the following properties:

Self-organization and Scalability DHT-based networks as CAN, Chord and Pastry self-organizing context for large-scale P2P networks. In all protocols, nodes and objects are signed random identifiers (called node identifiers and keys respectively) from a large, sparse id space. A route primitive forwards a message to the live node that is the closest in the id space to the message's key. Furthermore, given that each node has a well defined routing table, the lookup for any node/object can be accomplished within a relatively number of hops (Pastry in log16N, Chord in (1/2)log2N and CAN dN1/d, where N is the number of nodes in the overlay and d the dimension of space in CAN.

Efficiency P2P networks based in DHT-algorithms improve significantly the use of the available network bandwidth, in a trade-off for slightly more computation during query resolution. This was the case of the second version of the JAVA implementation of JXTA [Li, 2003].

3.5 P2P Models

Nowadays, there are several programming APIs and models to create P2P applications. These models define a set of abstractions required to represent a P2P system and also define a set of services, typically found in this type of networks. Two of these P2P models are described in this section.

3.5.1 JXTA

JXTA [Wilson, 2002] is an open-source-based P2P infrastructure developed at SUN Microsystems by Bill Joy and Mike Clary (2001). It is P2P library that implements a set of protocol specifications. In addition, JXTA identifies some concepts considered by JXTA team as relevant for P2P programming.

3.5.1.1 JXTA Elements

The elements that JXTA identify for P2P programming are the following:

Peer A Peer is a node in a P2P network that forms the fundamental processing unit of any solution. A node is any networked device (computers, PDAs, servers, printers, etc.) that implements one or more JXTA protocols. It is possible to have more than one peer running on the same device. There are no restrictions of operating and synchronization dependencies between peers. An encompassed definition of a peer is: *Any entity capable of performing some useful work and communicating the results of that work to another entity over a network, either directly or indirectly* [Wilson, 2002]. According to the type of work that a peer can have it can be:

Simple Peer , designed to serve to a single user, allowing to him to provide and consume services from other peers on the network.. Often this kind of peers are under strong security conditions (behind firewalls, NATs, etc.). They will probably not be capable of communicating with peers that are outside their barriers. That's why in the JXTA scheme they have the least responsibility in the P2P network.

Rendezvous Peer , it is a meeting place that provides peers with a network location to use to discover other peers and peers resources. Rendezvous peers are usually outside a private internal network's firewall. If it is necessary to locate it inside, either a protocol authorized by the firewall or a router peer are required.

Router Peer it provides a mechanism for peers to communicate with other peers separated from the network by firewall or NATs equipment. A router peer provides to peers outside the firewall a go-between that they use to communicate with a peer behind the firewall, and vice versa.

These types of roles are not exclusive.

Peer Group is a set of peers having a common interest. It provides services to their member peers. They determine its membership policies. The kind of common goals of Peer Groups can be based on:

- The application they want to collaborate on as a group
- The security requirements of the peers involved
- The need for status information on members of the group.

A Peer can be in several Peer groups simultaneously.

Network Transport There are mechanisms to allow peers to exchange data over the network. This layer, known as *network transport* [Wilson, 2002], is the responsible for all aspects of data transmission that can be breaking the data into manageable packets, adding appropriate headers to a packet and in some cases, ensuring that packets arrive at their destination.

Network Transport is composed by three parts:

1. Endpoints the initial source or final destination of any piece of data being transmitted over the network. An endpoint corresponds to network interfaces used to send and receive data.

2. Pipes unidirectional, asynchronous, virtual communications channels connecting two or more peers.

3. Messages containers for data being transmitted over a pipe from one peer to another.

Services Services are the *useful tasks* a Peer or Group Peer can perform. These tasks can be anything that a peer might want another peer in a P2P network to be capable of doing (like transferring a file, and so on). There are two categories:

1. Peer services offered by a peer to other peers. These services are available only when the peer is connected.

2. Peer Group services offered by a Peer Group to its members. This functionality could be provided by several members of the group. This way, a service is available when at least one of the peers offering this service is online. Hence it improves reliability.

Advertisement *A structured representation of an entity, service, or resource made available by a peer or peer group as part of a P2P network* [Wilson, 2002]. Language-neutral metadata structures (XML documents) are used to represent advertisements.

Entity Naming Most items on a P2P network need a unique identifier:

- Peers need it for allowing other peers to locate it.

- Peer groups have an identifier for allowing other peers to perform actions inside of it, such as joining, querying or leaving the group.

- Pipes use an identifier to allow communication between endpoints.

- Messages need it to be uniquely identifiable. This way peers can mirror contents across the network and provide redundant access, if possible.

3.5.1.2 Core JXTA Design Principles

The first design choice of the Project JXTA team was not to make assumptions about the type of operating system or development language employed by a peer. Furthermore the JXTA Protocols Specification expressly states that network peers may be any type of device. In addition, JXTA makes no assumptions about the network transport mechanism, except for a requirement that states that JXTA must not require broadcast or multicast transport capabilities.

3.5.1.3 JXTA Protocol Suite

Based on the protocols specification for any P2P system implementation, JXTA developed six protocols based on XML messages (figure 3.6). Each protocol conversation is divided into a portions conducted by the local and the remote peer. Each one is responsible for handling the incoming message and processing the message to perform a task.

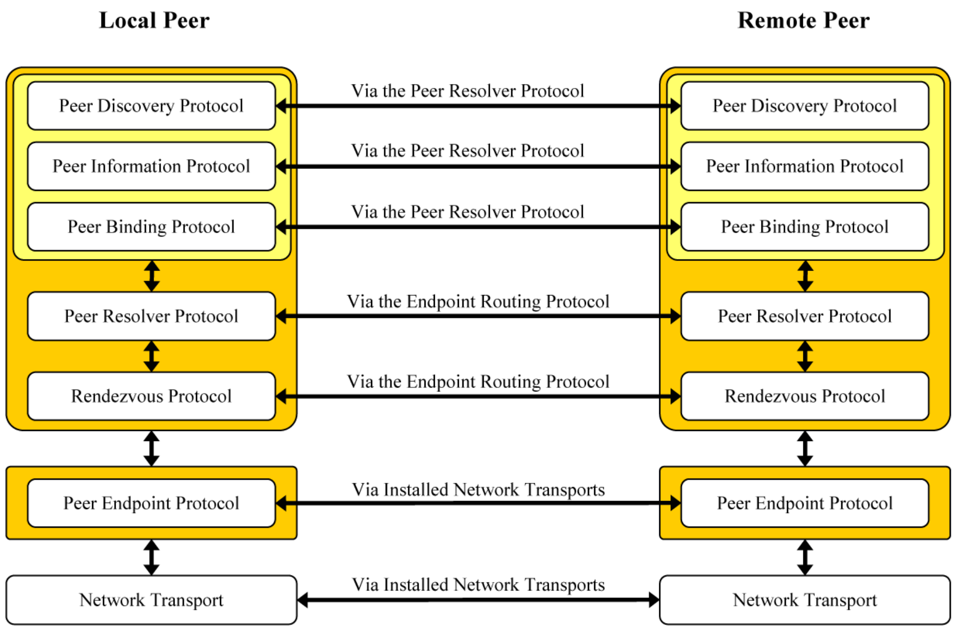

Local Peer
Remote Peer
Peer Discovery Protocol
Via the Peer Resolver Protocol
Peer Discovery Protocol
Peer Information Protocol
Via the Peer Resolver Protocol
Peer Information Protocol
Peer Binding Protocol
Via the Peer Resolver Protocol
Peer Binding Protocol
Peer Resolver Protocol
Via the Endpoint Routing Protocol
Peer Resolver Protocol
Rendezvous Protocol
Via the Endpoint Routing Protocol
Rendezvous Protocol
Peer Endpoint Protocol
Via Installed Network Transports
Peer Endpoint Protocol
Network Transport
Via Installed Network Transports
Network Transport

Figure 3.6: JXTA Protocol Stack

Protocols are not totally independent for one another because each layer in the JXTA protocol stack depends on the layer below to provide connectivity to other peers. However, peers can use a subset of all kind of protocols.

The Endpoint Routing Protocol provides a set of messages used to enable message routing from a source peer to a destination peer. Route information includes an ordered sequence of relay peer IDs that can be used to send a message to the destination.

The Rendezvous Protocol allows for peers to subscribe or be a subscriber to a propagation service. The former corresponds to a peer that is listening to a rendezvous peer. The latter is the rendezvous peer itself.

The Peer Resolver Protocol allows peers to send and process generic requests and receive an associated response. Queries and responses can be directed to all peers in a Peer Group or to specific peers within the group.

The Pipe Binding Protocol provides a mechanism to bind a virtual communication channel to a peer endpoint.

The Peer Information Protocol provides peers with a way to obtain status information from other peers on the network. This information can be uptime, state, recent traffic, etc.

The Peer Discovery Protocol allows for peers to advertise their own resources (e.g., peers, peer groups, pipes or services) and discover resources from other peers. Each peer resource is described and published using an advertisement.

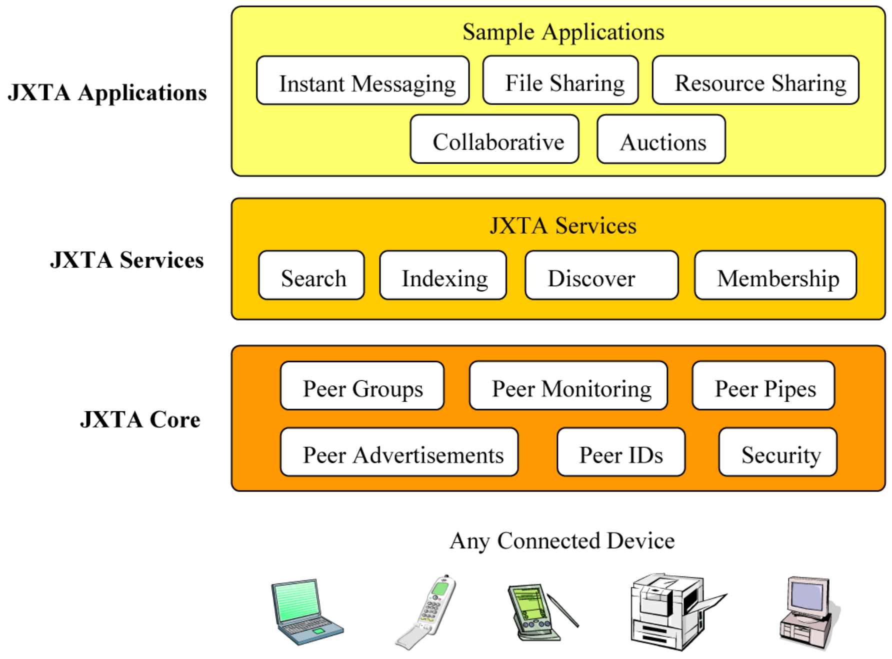


Figure 3.7: Project JXTA Software Architecture

3.5.1.4 Logical Layers of JXTA

JXTA platform has three layers (figure 3.7). Each one of them *builds on the capabilities of the layer below, adding functionality and behavioral complexity* [Wilson, 2002]:

Core Layer provides the essential elements of every P2P application, namely Peers, Peer Groups, Network Transport, Advertisements, Entity Naming, Protocols and Security and authentication primitives.

Services Layer it provides *desirable* P2P services such as searching for resources on a peer, sharing files from a peer and performing peer authentication.

Applications Layer it is built on the capabilities of the services layer to provide the ”common P2P applications”.

3.5.1.5 Evaluation of JXTA

JXTA succeeds in creating a standard protocol and a set of patterns of P2P services that is widely used. The protocol is independent of application, devices and network transports to be used in P2P systems. In addition, the use of XML provides a standard-based format for data structures which is largely understood, easily transportable (light weight) and human-readable.

Benchmarks have revealed that Java implementation of JXTA [Mathieu Jan, 2004, Emir Halepovic, 2003] suffer from reliability and performance problems (additional overheads and high latency values). However, they confirm that JXTA is anyway an appropriate solution for common challenges of P2P network, such as large data transfers or slower-speed networks.

Finally, what apparently is the major advantage of JXTA (to be a standard protocol) could become the biggest limit. It has already made a number of design decisions. This could confine the protocol to specific type of P2P applications. JXTA has to have the definition of its protocols flexible enough to adapt itself to new types of application in the future or combination of existing P2P solutions in order to extract the best of each of them to future systems. These kind of combined approaches have already started to happen [Bernard Traversat and Pouyoul, 2003].

After having worked with JXTA one may to conclude that this model could be even much simpler, in the sense the JXTA concepts seem to be more abstractions of elements found in a programming language (like classes) than elements found in a conceptual P2P system . A case that shows this difference is the *pipe* concept. It is clear that a pipe is an excellent abstraction of a socket, but is this concept indispensable to design a P2P system? What is really essential is the possibility of a peer to send a message to another peer. It could be represented just as a method.

3.5.2 P2PS

P2PS is a library/platform to build P2P application in Mozart/Oz created at Université Catholique de Louvain, Belgium in 2003. P2PS implements Tango algorithm [Carton and Mesaros, 2004], which is a generalization of Chord but it scales better, in order to get efficient development. The *P2PS platform provides the developer with a means for building and working with P2P overlay applications, offering different primitives and services such as group communication, efficient data location, and dealing with highly dynamic networks.* [Valentin Mesaros and Roy, 2004]

3.5.2.1 Functionality

P2PS has an API that offers the primitives and services hiding underlying details such as the DHT algorithm used (whereas it implements Tango; it can be any DHT algorithm). The primitives are used for management, communication and monitoring.

Management Primitives

Create a Network functionality that allows user to create a P2P overlay network. It creates the first peer node of a network; specifically it creates the access point for this node. The primitive is *createNet* and receive parameters related to the network, to the node and to the access point (IP and port). Finally the access point is published in the network.

Join a Network Primitive *joinNet* that receives as parameters the access point of a peer node already joined to this network.

Leave a Network Primitive *leaveNet* that disconnect the peer node from the network.

Communication Primitives P2PS performs efficient key based routing that means a message from a node s to a node d will be routed throughout the overlay network according with the corresponding key lookup procedure, where d is considered a key.

One-to-one communication primitive to send a message. It is possible to choose between sending the message *directly* or *indirectly* (through the responsible of the key), and doing the *best-effort* or *reliably*.

One-to-many communication primitive to send a message to all or a selected list of peer nodes in the network.

Send to successor method to get replicas of the message in more than one node. It is done to increase the resilience.

Monitoring Primitives Primitives to be aware of the state and changes respect to peer nodes and network.

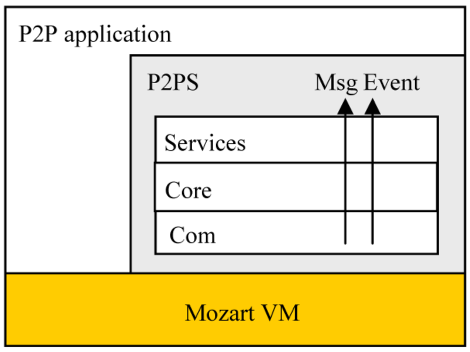


Figure 3.8: P2PS Architecture

3.5.2.2 Architecture

P2PS is organized in three-layer architecture (see figure 3.8).

Com Layer Layer that act as the interface of P2PS and the underlying physical network. This layer provides the functionalities of access point creation, connection establishment, basic communication primitives and fault detection.

Core Layer Layer that provides high-level connectivity primitives. It implements the Tango algorithm. Its main purpose is to *implement node join and leave mechanisms, route key based messages to their responsible, and maintain the routing table and the successor list regardless of the nodes joining and leaving, thus guaranteeing overlay efficiency.* [Valentin Mesaros and Roy, 2004]

Services Layer This layer is a kind of wrapper that builds up the operations needed to implement P2P applications. These are system initialization, create connection access and systems join and leave operation.

3.5.2.3 Evaluation of P2PS

This model results much simpler than JXTA. As Mozart/Oz, P2PS is a good model because of its expressiveness. Even while JXTA can offer more services, such as security and authority, P2PS reduce all its services to a small set of primitives, whose counterpart in JXTA comprises a considerable sequence of object instantiations and explicit subscription to these services.

3.6 Open P2P Networks in Mobile Environments

The previous sections described the evolution of P2P network architectures in the context of the internet. As said before, this is an open network in the sense it has a dynamic configuration due to the connection volatility of their participants. P2P architectures have demonstrated to work properly under these dynamic conditions.

Recently experiences [Ding and Bhargava, 2003, J. Kurhinen and Vuori, 2004, Priyantha et al., 2000] have used P2P architectures in the context of mobile networks. Particularly in the context of mobile ad-hoc networks explained in the previous chapter (section 2.2.2.2). Some similarities and differences between both phenomenons have been found in these experiences. The similarities are the following:

- Neither mobile ad-hoc networks nor P2P network architecture require a central server, which means, every peer or device should collaborate with others in order to make the whole system work.
- One of the main problems in P2P architectures and mobile ad-hoc networks is to discover peers and services over the network, and routing communications efficiently.
- In both systems, broadcasting can be employed to some extent in order to exchange data or routing information among different peers. As was explained in this chapter, such a strategy can raise the scalability problem.
- There is a mechanism in mobile ad-hoc networks called proactive (or table-driven) routing protocols in which every mobile node tries to maintain a routing table involving the complete information of network topology. This protocol is similar to the routing mechanisms of DHT-based P2P architectures.

The inclusion of P2P architectures in mobile networks is just starting to produce results. There are still some issues to solve, such as the following:

- Mobile ad-hoc networks and P2P systems work in different network layers. There are no clarity respect to the coordination mechanisms that should exist between both systems.
- Mobile networks are constrained by limited resources of theirs participants. It is not a concern of P2P applications over internet (such as file sharing systems).

- Whereas some P2P systems on internet would be interested to establish the same connections after a disconnection, mobile ad-hoc networks focused to reestablish the connection independently of the channel used for this purpose.

3.7 Conclusion

This work has described the history of P2P systems since the creation of the internet. During this history networks (both software and hardware) have evolved according to several interests. Because of these interests the P2P paradigm was left out for a long time. But the same evolution and the new *sophisticated* user requirements such as sharing files or communicating over the internet, brought back the P2P systems. These requirements do not differ so much of the requirement that the early applications' developers imagined at the very beginning.

New kinds of P2P applications, algorithms and models have been evaluated in this chapter. As a summary it is possible to say:

- P2P applications are decentralized, self-organized, highly scalable and fault-tolerant. On the other hand these applications require decentralized coordination, which is more difficult to handle than its centralized counterpart; are more sensitive to the peer conditions; and finally could imply more programming issues.

- Current P2P applications can be classified as centralized, decentralized and hybrid. This difference is based in where and how services are provided. Hybrid P2P systems with the incorporation of a second type of peer (with better conditions) have gotten the best test results.

- P2P systems based in DHT algorithms get better performance in all works related to peer lookups due to its adequate infrastructure for this service of peer discovery.

- P2P models and APIs identify a set of entities and services for P2P application. Users can create their own P2P systems based in those entities and services.

- Finally, this work have described recently experiences implementing P2P architectures on mobile systems. This architecture provides some properties that comply with the requirements of mobile networks (since they are open networks as the internet). However this type of experiences have just started, and several issues remain.

The following chapter will present the ambient-oriented programming paradigm. This paradigm aims to solve the problems found in mobile networks mentioned in the chapter 2. It also requires decentralized systems such as the systems developed with P2P architectures. As will be explained, AmbientTalk [Dedecker, 2005a] is a language that implements this paradigm and provides P2P solutions to deal with mobile devices.

Chapter 4

The Ambient-Oriented Programming

4.1 Introduction

The particular interest of this chapter is to illustrate the way in which current concurrency and distribution models deal with the new conditions of mobile networks. Most of these approaches are adaptations to past models developed for other types of networks. They are briefly defined in this chapter discussing their applicability in this new context.

One of the arguments presented in this chapter is that the concurrency model known as *Actor model* [Agha, 1986] has more adequate properties to deal with mobile open networks than other traditional approaches. However, this model is still not good enough to comply all requirements imposed by mobile environments. Remember that the special condition of mobile open networks, explained in chapter 2, is their dynamic configuration due mainly to the mobility of their participants.

This chapter presents the actor-based concurrency model called *Ambient Actor model*. This model was developed in the context of the ambient-oriented programming language called *AmbientTalk* [Dedecker, 2005a]. It deals with the conditions of mobile open networks, giving actors the ability to foresee changes in the environment, so that they can take appropriate actions.

Several programming languages and frameworks have adapted the actor model to work with different networks. The current approaches that use this model to deal with mobile networks are somehow based on previous implementations done for other contexts. This chapter identifies the links between the different actor-based languages and frameworks.

4.2 Open P2P Networks Conditions for Concurrency

Concurrency appears when two or more computations work together. In a mobile open network these computations can be interacting from different devices. Such a coordinated work should consider the properties of open networks and their participants (devices) described in the chapter 2. Some of this properties can affect the way in which devices work together. The conditions these properties hint are the following:

1. Devices are concurrent by nature. All the devices that work together over a open network are running independently (each one has its own processor). As such, a concurrent work would not must affect this autonomicity, but it could take advantage of this property.

2. Mobile devices have a volatile connection and scarce resources. A concurrent system should minimize the impact of these limitations.

3. A mobile system would require to move computations from one device to another (some reasons are described in the chapter 5 about mobility). The concurrent work should not hamper this mobility.

The following section describes different approaches to model concurrent distributed systems and discusses the issues found in these approaches to model systems for open networks.

4.3 Concurrency Models

This section introduces two concurrency models, one based on threads and another in actors. Although there is not a *thread model* as such, there are many programming languages that deal with concurrency by implementing threads. To simplify the identification of this approach, it will be referred to the *thread model* in the context of this work.

[Briot et al., 1998] defines a three-dimensional design space that classify the implementations of concurrency models in programming languages. The dimensions are the followings:

Alignment between objects and threads This dimension *reflects* the degree of alignment between objects (as state containers) and threads (as independent processors) are. The entities with a full alignment of these two concepts are known as *active objects* or *actors*. These are objects that contain their own execution threads.

Alignment between messages and synchronization boundaries This dimension reflects in what degree the messages communicated between objects produce synchronization in their processes. In other terms, it measures to which extends an object that sends a message to another, gets blocked until the reception of the response.

Alignment between objects and units of distribution This dimension classifies the languages according to the fact that object and unit of distribution are separated or not. When an object is the unit of distribution, objects as a whole are the only remotely accessible data, and an object always conceptually resides at one node.

These three dimensions will help to compare the thread and actor models in the two following sections.

4.3.1 Thread Model

A thread is a virtual processor capable to execute instructions sequentially, and communicate with other threads by sharing data. In an object-oriented language, each thread can execute methods of multiple objects. It reveals that threads and objects are totally unaligned entities[1].

This model has well-known drawbacks. These are the following:

Data-races A data-race can occur when a shared data is manipulated simultaneously by two threads. It can lead the data-structure to an inconsistent state (for instance, if the `put` and `get` methods of buffer are called at the same time). A program with data-race problems can randomly crash, give incorrect answers and be very hard to debug.

Deadlocks A deadlock occurs when a set of threads interfere in such a way that all of them get blocked waiting for a resource they will never acquire. It produces a cyclic dependency between threads such that each thread holds a resource which its successor in the cycle is waiting to acquire.

Starvation Starvation appears when a thread never acquires a resource because it is systematically granted to other threads. It also can happen after the application of mutual exclusion schemes. Although this problem can be *less critical* than the previous ones, this is another case making evident the *extra* complexity of languages that let on programmers the responsibility to deal with concurrency.

The next section will explain the way in which languages implementing the thread model, deal with these drawbacks.

4.3.1.1 Thread-based Languages

Many class-based languages deal with concurrency by using a thread model. Java's concurrency model [SUN Microsystems, 2005] considers two other concepts in addition to threads: monitors and synchronization [Lea, 1999]. Both concepts have

[1]Some class-based languages implementing the thread model, like Java, define a thread as a class too. However, the instances of this class are quite unlike active objects.

been included to deal with the drawbacks mentioned above. Java threads, monitors and the synchronization mechanism operate as follows:

Java threads In order to specify the behavior of a thread, an object should be an instance of a class that implements an interface called `Runnable` (with one method called `run`). This interface is already implemented by a Thread class. In such a case, the processor executing the thread will start when the `start()` method of this class is invoked, and will finish whenever the `run` method returns.

Synchronization Java programmers have to deal with synchronization to avoid data-races. Programmers are responsible of ensuring the *mutual exclusion* of threads by declaring *critical* (shared) pieces of code as `synchronized`[2]. Thus, only one object will execute such a critical section. The rest of the threads in the system trying to access this section must wait until the current thread finishes its work. Note that a deadlock can be considered the side-effect of Java synchronization, since it occurs only when the threads share resources under mutual exclusion schemes.

Monitors A monitor encapsulates mutable data shared between multiple threads, by means of a data structure mediating the access to this data. In Java every object is a monitor that can be requested and released. Thus, a `synchronized` block always is referring to a monitor. Sometimes an operation cannot be executed while a condition is not verified (it is related to the *conditional* or *behavioral* synchronization [De Meuter, 2004], which is explained in the section 4.6.2.1). In such a case, a thread can be suspended until the condition is fulfilled. In Java this is achieved by sending the message `wait` to the monitor of the object containing the synchronized code. This releases in that particular moment usually to allow other threads access to execute this critical code. To notify that a condition may be verified, the monitor can receive a message `notifyAll`. At this moment all waiting threads are resumed.

4.3.1.2 Evaluation of Thread Model

It is to be said that all the drawbacks of the thread model can be solved by using some patterns when coding concurrent programs. However, note that the problems were not produced by the concurrency mechanism itself (all Java concurrency constructs work properly), but the explicit incorporation of these mechanisms at programming level. For the following model based on actors, concurrency is transparent for the programmers.

With respect to the three design dimensions of [Briot et al., 1998], Java objects are neither aligned to a thread, nor to a unit of distribution. Even when an object could behave as a thread by implementing the `Runnable` interface and an object

[2]In Java it is a prefix in a method definition or a block used for synchronizing a part of a method body.

could be transmitted and identified over the network (using Java serialization and Java RMI mechanisms), such alignments (if any) must be manually. There are several programming languages explained in the section where objects are inherently processors and unit of distribution (in such cases programmers never have to align these concepts manually).

Finally, with respect to the alignment between objects and synchronization boundaries, Java uses two mechanisms of synchronization. The first mechanism, described above, is created to avoid race-conditions inside of the state of the object. The second one is the synchronous message sending mechanism used for communication with a single thread between objects. The programming languages in section present concurrency and distribution model that work properly without considering any of these type of synchronization.

4.3.2 Actor Model

The Actor Model [Agha, 1986] was proposed by Gul Agha at University of Illinois in 1986. This model is based on the concept of *Actor* introduced by Carl Hewitt at MIT in the early 1970s. Agha defines actors as *self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing*. An actor can perform three basic actions:

Sending Messages Actors communicate with one another using asynchronous messages. In other words, an actor that sends a message to another does not have to wait until the receiver actor processes the message and sends the response (if any) back to the sender actor.

Creating Actors To create an actor with a specified behavior description.

Become To allow an actor to change its own behavior. According to Agha, behaviors are *absolute containments of information* that can be shared (accessed and modified) in such a way that deadlocks in communications are avoided (however there can still be a sort of *livelocks* described in section 4.3.2.2).

An actor is a computation that encapsulates a *state*, a set of procedures controlling this state, known as its *behavior* and a execution *thread*. The actor communicates with others by an asynchronous message passing mechanism. For this purpose an actor has also a queue, commonly known as its *mailbox*, in order to receive the incoming messages. This message queue simplifies the concurrency by processing one message at a time, hence excluding intra-object concurrency (external threads entering to the state of the actor) such that race conditions on the internal state of an object are avoided.

According to Agha actors are defined to have a *history-sensitive behavior* which always can be expressed as a function of the incoming communications (messages). An actor will autonomously reacts to messages by executing the method body itself. Each method in an actor's behavior should specify the replacement of the current one that will be used to process the next message (see figure 4.1).

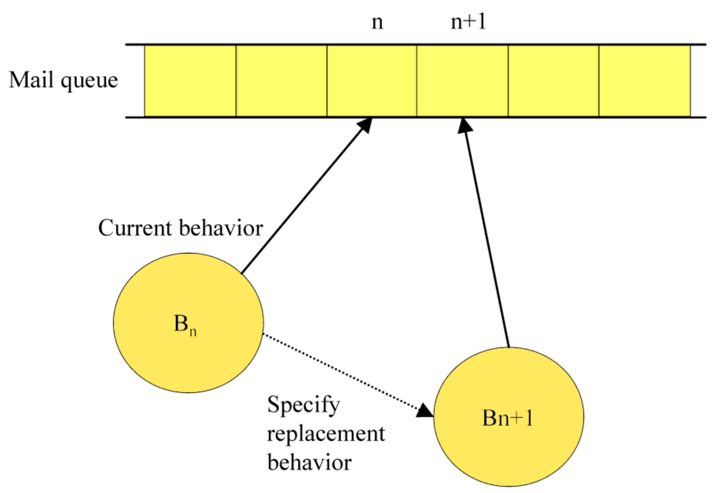


Figure 4.1: Actor Model

Actors are also defined to avoid sequentiality in the messages communication and processing. The first kind of sequentiality is achieved by using asynchronous message-passing. The second kind of sequentiality is achieved by including the actor creation process as part of the model. This action included in any method body allow actors to create new actors guaranteing the ability of increasing the distribution of a computation as it evolves.

4.3.2.1 Promise or Future Pipelining

As said before, actors have an inherent asynchronous communication system. Nevertheless, some actor-based languages provide mechanisms to allow synchronized sequences of actions among actors written in a *continuation-passing*[3] style. This mechanism usually includes messages with an extra argument containing an actor called *continuation actor*[Agha, 1986] or *customer*. When an actor receives a message, it executes the method corresponding to this message. The return value (called *join continuation* in this context) of the method is sent to the continuation actor. It will *consume* the join continuation such that the computation can proceed.

[3]*A continuation actually denotes "all that remains to be computed" at the point where the continuation is captured. Continuations are best understood when looking at programs as nested expressions, each of which "returns its value" to the surrounding expression, when it gets evaluated. This "expression surrounding the value" is then the continuation of the expression yielding that value.*[Van Custem and Mostinckx, 2004]

It is possible to find continuation chains as result of sequential computations. In such a scenario, computations are composed by a sequence of method invocations. Each invocation uses the result returned by the previous one as a parameter. In conventional remote procedure call (RPC) systems, it would require many synchronous *round trips* [Miller, 2000] over the network, one for each method invocations. For languages using continuations, it is nothing more but a sequence of messages sent from an actor to another. Every message uses the continuation actor (also called *promise* [Miller, 2000, Liskov, 1988] or *future* [R. H. Halstead, 1985] in this context) returned by the receiver actor of the previous message, as parameter for the next message.

An example of this mechanism can be found in the *when-catch* construct of E programming language [Miller, 2000] (see the code below). It can be read as follows:

> *When a promise (continuation) becomes done, and therefore the resultant object (`temp`) is locally available, perform the main action block. But if something goes wrong, catch the problem in variable prob and perform the problem block [Miller, 2000].*

```
# E syntax when (tempVow) -> done (temp) {
    #... use temp
} catch prob {
    #... report problem
} finally {
    #... log event
}
```

The section 4.6.3.1 will show that the continuation pipelining mechanism is quite useful for network reconfigurability purposes in open networks.

4.3.2.2 Evaluation of Actor Model

Actor model offers an expressive and transparent alternative for dealing with concurrency and distribution. Three primitives are expressive enough to represent scenarios free of the common concurrency problems such as deadlocks, data-races and starvation (although the actor paradigm does not impose an order in the message execution it can be assumed the same order of the reception of the messages). Actor-based languages such as the ones described in the section 4.5 keep programmers away from low-level implementation details of concurrency and distribution.

In terms of the three-dimensional design space explained at the beginning of this chapter, actor-based languages align the object, the thread and the unit of distribution (all of them are contained in its computation unit called *actor*). In addition, messages between actors do not compromise the autonomy of each of them. It is also convenient for systems running on dynamically reconfigurable environments like open networks, since synchronized communications between actors make them more vulnerable to the connection volatility of the device where they reside. Nevertheless, it does not mean that there is no synchronization at all. De Meuter [De Meuter, 2004] identifies a type of synchronization related to actor behaviors. It will be explained in section 4.6.2.1.

This model is not free of problems. There are still issues to be solved or improved. For instance:

- The programs using continuations can become *scattered* and *unreadable*. As explained in 4.3.2.1, it is because *the breakdown of sequentiality in a computation is also a breakdown of sequentiality at level of the implementation of method's bodies*.
- The continuations and continuations chains can impose an overhead on the system, since it is a mechanism that only postpones the work that at some moment will have to be executed.
- As important as the expressiveness is in communication, is the properly reception of the messages and its responses, if any. In communications between local actors it can be matter of a good implementation of the methods in behaviors, but in a distributed system it also has to do with the availability of both sender and receiver in the network during the communication. This fact is more evident when working with mobile open networks (see section 4.7). In such a case an actor can send a message to another and time after, when the receiver actor produces the response, the sender actor can then be out of range. More considerations related to concurrency models for open P2P networks are exposed in the following section.
- The actor model avoid deadlocks but there can still be a sort of ”livelock”: messages can be ”stuck” in the incoming message queue of an actor if it does not have the appropriate behavior to execute the message. It does not block the actor, but it may stall the progress of an application.

4.4 Concurrency Models' Issues for Open P2P networks

There is a set of issues identified in [De Meuter, 2004] when dealing with concurrency in open P2P networks. Some of them are directly related with the conditions of this type of network described in section 4.2. The issues are the following:

1. Using synchronous message sending would block unnecessarily the devices. Asynchronous communications solve this problem. The same argument is useful for communications that involve more than two actors. The promise pipelining mechanism minimizes the coupling between devices and can be used for network reconfigurability purposes (see section 4.6.3.1).

2. As said in the previous section 4.3.2.2, languages implementing continuation passing style can be complex to code. The problem is even worse in open P2P networks in which devices can get out of range during the communication process, forcing programmers to think *how to keep the connection alive* during the process. However, resource limitations in the current devices and networks, make impossible to ensure the full communication reliability in the context of open networks. Some mechanisms have been proposed to give alternative solutions to this problem (see section 4.7.1.1).

3. The model has to facilitate the work with mobile setting where actors hop from one device to another. The chapter about mobility (chapter 5) will explain why the actor model is more convenient for mobility purposes than a thread-based one.

4. The difficulties to work with threads that share state, motivated researchers to create inherently concurrent actors. However, in the context of open networks it is possible to find cooperating distributed applications that need a kind of *shared state* (acquaintance) between the concurrently operating devices (like shared virtual white boards). In pure actor systems this is hard to accomplish by sending messages back and forth all the time, using explicitly *encoded session information* (such as http cookies). De Meuter suggests letting participants to share some state. Section 4.5.2 describes ABCL, a programming language that implements the actor model including a notion of a actor states. However it is unaccessible by external actors.

5. It is hard to guarantee consistency of internal actor state in open networks if it allows intra-object concurrency. This is due to the inability to foresee all interactions this actor can undergo with actors in other devices.

4.5 Actor-based Languages and Frameworks

The following sections boils down to two early actor-based approaches. The first one corresponds to an ambitious idea to use the properties of actors in *intelligent* entities known as *Agents*. The second one is an actor-based programming language called ABCL, although it was not conceived for open P2P network (as the languages presented in section 4.6), it has become a unavoidable reference for later actor-based languages.

4.5.1 Actor as Agents

The concept of agent has been present since very early times in technology [Lange and Oshima, 1998], however there is still no agreement about its definition. An agent can be loosely defined as a *software entity that assists people by performing tasks under given constraints on their behalf* [Clements et al., 1997]. The ambitious idea behind this concept is to provide this agent with intelligence.

But this intelligence has been related along the history to many attributes like reactivity, autonomy, collaborative behavior, knowledge-level communication ability, inferential capability, adaptability, mobility, and others. In a limited consensus, some authors[Clements et al., 1997] say that an agent must accomplish at least the following conditions:

Goal-oriented An agent should act in response to changes in their environment related to a goal.

Communicative An agent should be able to communicate with other agents and with its environment.

Mobile An agent should be able to transport itself from one host to another.

The reason why agents are considered as actors is because usually the agents communicate with each other by message-passing, like actors do.

4.5.1.1 Aglets

There have been a large number of agent building packages. One of the most popular is the Aglets Workbench[Lange and Oshima, 1998], developed at IBM's research labs in Japan (1998). An aglet is a java-based onternet agent (*Aglet* is a shorthand for agent plus applet) characterized by an event-driven entity (as a Java applet). These events provide an aglet with state persistency and light-weight object migration mechanisms. The aglets reside in servers containing objects called *contexts*, which act as workplaces in which aglets can communicate between them and get information about the server. This information is used by a aglet to take its decisions (for instance, to check the compatibility with a certain server it intends to migrate to).

Concurrency Model of Aglets As said previously, aglets use a message-passing communication scheme. They do not normally invoke each other's methods directly. Instead, they send messages through proxy objects obtained from the server known as `AgletProxy`, in the same line of Java RMI-based systems. An aglet can send both synchronous and asynchronous message to another by using the proxy methods `sendMessage` or `sendAsynchMessage` respectively. Each message carries a `String` indicating the kind of message and another optional piece of data. A particular property of aglets is that they can *comply*, *refuse to comply* or *decide to comply later with* a request. This is possible thanks to the *callback* model implemented in the aglets framework [Venners, 1997], which provides them with an instant of decision (the *callback* method) previous to the execution of any event. For instance, the `dispatch()` method (used to move aglets) is invoked on an aglet, the `onDispatch()` *callback* method is invoked by the host, before the dispatch operation starts. Therefore, in the body of `onDispatch()` the aglet must decide whether or not go.

4.5.1.2 Evaluation of Aglets

The Aglets technology is strongly based on a client-server architecture. As explained in chapter 3, it is not the most appropriated scheme for open networks. On another hand, the close relationship between an aglet and its server containing it, implies that the latter commonly has to be referred at programming level (for instance, to get an AgletProxy of another aglet).

4.5.2 ABCL

Actor Based Concurrent Language[Briot and de Ratuld, 1988] (acronym ABCL) is a prototyped-based concurrent programming language created by Akinori Yonezawa et al. at University of Tokyo in 1986. It provides active objects (with an associated execution thread).

4.5.2.1 Concurrency Model of ABCL

ABCL implements the actor model and combines it with the notion of *mutable state*[De Meuter, 2004] which can be *dormant*, *active* or *waiting*. An active object is initially *dormant* and becomes *active* whenever it receives a message. It becomes *dormant* again when there are no more messages in the message queue. Active objects can get blocked waiting for a certain message. This is achieved using a mechanism known as *selective message receipt*[De Meuter, 2004] which is incorporated in ABCL by the `select` construct.

In order to protect active object states from concurrency problems related to state sharing described in section 4.3.1.2, ABCL takes the following considerations:

- Active objects cannot process more than one message at time.
- Methods patterns can contains additional *constraints*[De Meuter, 2004] (for the behavioral synchronization explained in section 4.6.2.1). A same method pattern can occur with multiple different constraints.
- If multiple pattern-constraint pairs match an incoming message, the first one is selected.

Messages sent to an active object are put in its message queue (ordinary at the end) if the object is in *active* state. If it is in *waiting* state, it will check whether the incoming message is the expected one, according to the required patterns.

ABCL features three kinds of messages: *past*, *now* and *future*.

Past Type Message Passing This is a pure actor-like asynchronous method invocation, the sender object does not wait for the response.

Now Type Message Passing It is equivalent to the conventional synchronous method invocation. In such a case, sender has to wait for a result to be returned. The difference is that the receiver object can carry out some computation after sending back the result (since this is an active object with an independent processor). Recursively calling a method through now type message passing causes deadlocks [De Meuter, 2004].

Future Type Message Passing This type implements the promise pipelining model described in section 4.3.2.1. Asynchronous method calls return a result (a *future* object) which can be used as a parameter for a next computation. Unlike Multilisp's futures[R. H. Halstead, 1985] or Argus' promises[Liskov, 1988], ABCL allows a computation to fulfill more than once. To support this, the future object has a queue that accumulates all values returned.

There is an alternative message passing mode called *express* mode, in which *express* messages are accepted even when the receiver object is *active*. This mode implies message-handling to lose its atomicity. Therefore ABCL includes the primitive `atomic` to evaluate expressions without interruption.

4.5.2.2 Evaluation of ABCL

As mentioned above, ABCL has been an important reference for more recent actor-based languages, like ChitChat [De Meuter, 2004] and SALSA [Varela and Agha, 2001]. It is a prototype-based concurrent language based on active objects that provides three types of messaging passing. The first type (past) is the implementation of the *send* action in the original actor model. The second one is a improved version of the synchronous message sending found in thread-based languages. Although it still can produce deadlocks, the processing of message is done by the active object that contains this method, and not by the sender of the message (it avoids data-races). The last type works with future objects.

4.6 Actor-based Languages for Open P2P Networks

The purpose of this section is to present programming languages that deal with distribution and concurrency in open networks. All of these approaches consider somehow the three concepts reviewed until the moment in this work (open networks, P2P network architectures and actors). These properties will be highlighted throughout this section.

As a consequence of working with open networks, the following languages also provide mobility mechanisms that will be described in the next chapter exclusively dedicated to mobility.

Before describing some actor-based programming languages it will be discussed different approaches in which the actor model is implemented as a software library for a object-oriented language.

4.6.1 Actor-based Libraries

Several actor-based libraries have been implemented in different object-oriented languages. Some examples of these libraries are ProActive[Baude et al., 2003] and TAPAS[Shiaa and Aagesen, 2002]. Such libraries provide high-level middleware services for discovery, communication, and mobility.

MicroTAPAS is an interesting library to describe in the context of this work because it is an actor-based approach oriented towards mobility in open networks, providing some services in a P2P-style.

4.6.1.1 MicroTAPAS

MicroTAPAS[Luhr, 2004] was developed by Eirik Lühr at NTNU, Norway in 2003. It is a lightweight version of TAPAS[Shiaa and Aagesen, 2002] geared towards wireless environments. TAPAS (acronym for Telematics Architecture for Plug-and-play Systems) is a software architecture developed for facilitating the management of distributed services in a network. It allows for example for as the dynamic introduction of new services or the upgrade of them. MicroTAPAS strives for the same goal but now considering the use of small handled wireless devices. One of the main motivation for MicroTAPAS is to allow users to move around the network without the need of manually reconfiguring applications and services after the movement.

The current implementation of this library was built using Java 2 Micro Edition [SUN Microsystems, 2005], a reduced Java version for mobile applications.

MicroTAPAS Extended Actor Model This model is an extension of the Agha's actor model, considering *actors* as software components that reside in *nodes* of a network that can be servers, routers and switches, and users terminals, such as mobile phones, laptops, PCs, PDAs, etc. In addition, this model consider an extended actor composition and the creation of a *theater* metaphor, defined below.

Extended Actor In addition to the state, behavior, a queue of messages and an own execution thread, MicroTAPAS provides the actors with the following two components:

- **Capabilities** These are the requirements an actor demands of its current node (or future one in case of migration). In the scheme proposed in MicroTAPAS, *a capability required by an actor should be matched with a capability offered by the target node*[Luhr, 2004]. Capabilities can be hardware resources connected to the node (like printers or screens), but these can also be quantitative aspects such as *processing capacity* or *screen resolution*.
- **Role-sessions** These capture the relationships between actors needed to fulfill their services. It can even imply the creation of new actors if it is required.

Figure 4.2 shows the TAPAS extended actor model. How the components of an actor are affected by actor mobility will be explained in the next chapter related to mobility).

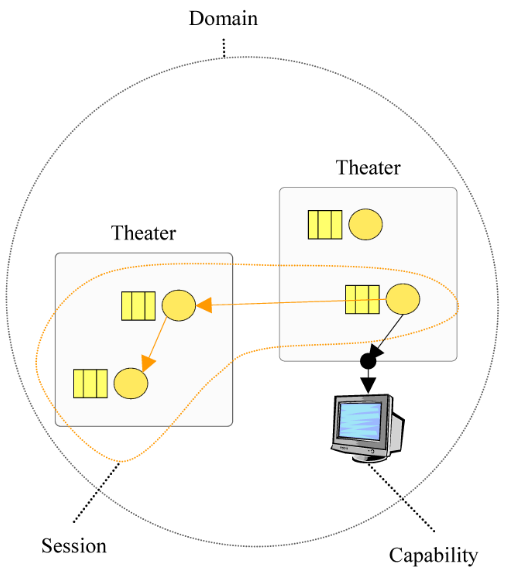


Figure 4.2: MicroTAPAS Extended Actor Model

Theater metaphor MicroTAPAS actor model is supplemented by a the theater methaphor[4]. Without going into too much details, this metaphor is composed by the following parts:

- *Actors* that perform *Roles* according to predefined behaviors.
- *Plays* consisting of several *Actors* with different *Roles* that are logically related.
- A *Director* acting as the manager of the *Plays* and supervisor of the *Actors*.
- A *Domain* that represents the population of *Actors* and *Nodes* managed by a *Director*.
- A *Configuration Manager*, who is the responsible for obtaining a snapshot of all system resources, and taking decisions about *Capability* and *Actor* installations.

[4]Several extensions to the actor model have adopted a ”Broadway” manner of naming all concepts used in the new models [Shiaa and Aagesen, 2002, Varela and Agha, 2001]. It allowed to extend the original model in such a way an *Actor* was still a meaningful concept in a more general metaphor that could include theaters, directors, managers, plays, etc.

- A *Mobility Manager* to manage the *Actors* and *Nodes* connectivity and mobility.

As mentioned before, most of the concepts are related to reconfigurability and migration tasks. The migration process and the participation of these components in it will be explained in the next chapter.

4.6.1.2 Evaluation of Actor-based Libraries

A library providing a good API (in terms of understandability) could look pretty similar to a language providing new instructions for the same purposes of the library. However, the latter has advantages, as the ones suggested in [Varela, 2001]. It identifies the following advantages to use actor-based programming language instead of actor frameworks:

Semantic constraints There are desired semantic properties that can be guaranteed at language level, such as the complete encapsulation of data and processing within an actor (to prevent multiple threads from mutating shared state).

API evolution Generating code from an actor language ensures that proper interfaces are always used to create and communicate with actors. This code will not be affected by the changes to the actors API.

Programmability Using an actor language improves the readability of developed programs. Developing in frameworks often implies using constructs of the language in which the framework was developed, to simulate actor operations. It can be confusing for programmers to relate actor semantics with the syntax of the language. For instance, actor creation or messaging sending are represented in a framework as (synchronous) method invocations.

These advantages can be identified when comparing MicroTAPAS with the following languages. Yet, it is to be remarked that MicroTAPAS framework makes actors adaptable to their environment, and it happens transparently for programmers using this library.

Last but not least, this MicroTAPAS framework still provides some services from a central server. These services are related to the *director* responsibilities such as the coordination of *capabilities* and *plays* in its domain. For each change in the conditions of the capabilities and plays, as well as the movements of the actors, it is needed in some way.

4.6.2 ChitChat

ChitChat is a prototyped-based programming language for open networks created by Wolfgang De Meuter at Vrije Universiteit Brussel in 2004 [De Meuter, 2004]. It is strongly based on the Pic% programming language [De Meuter et al., 2004].

ChitChat adapted it to support distribution, concurrency and mobility in open networks. As an extension of Pic%, ChitChat is a *classless object-oriented programming language with the full power of prototype-based languages*[De Meuter, 2004]. Its concurrency and distribution model are results of the *intersection* of properties found Emerald [Jul et al., 1988], Argus [Liskov, 1988], Obliq [Cardelli, 1995] and ABCL [Briot and de Ratuld, 1988].

4.6.2.1 Concurrency in ChitChat

The ChitChat concurrency model is inspired by ABCL, which is based on active-objects. A ChitChat active object is conceived as the combination of a *passive object* defining the behavior, a message queue and a thread that consumes the incoming messages sequentially. There is no intra-object concurrency in ChitChat, which means that inside of an active object only one method can be executed at time. Figure 4.3 depicts an active object in ChitChat.

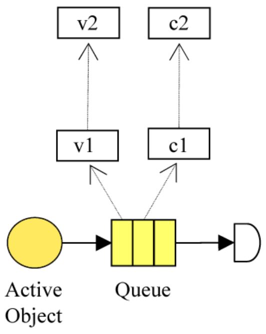


Figure 4.3: ChitChat Active Object

Containment Principle In ChitChat object structure complies the following principle: Each active object contains a *passive part* represented by passive objects used only within the boundaries of an active object. All references to passive objects never cross the network boundaries represented by the device in which these references are contained. If it is required, a deep copy of the passive object itself is passed on.

As a corollary of this principle only active objects can extend from each other over a network.

Behavioral Synchronization in ChitChat [De Meuter, 2004] describes behavioral synchronization (also called conditional synchronization) as making object operations temporally enable or disable depending on its internal state. ChitChat handles this by using a technique known as *call with current promise*[De Meuter, 2004]. The interpreter has always a *current promise* at hand, to wit the promise of the last sent asynchronous message. *Active objects are capable of grabbing the promise they are about to fulfill, store it and fulfill the promise manually, at a later moment, possibly after some conditions have been met*[De Meuter, 2004].

Active Objects and Delegation-based onheritance ChitChat uses delegation-based inheritance scheme to work with active objects. Thus, *an active parent can be shared by many active descendants*[De Meuter, 2004]. Active parents are intended to be the representative state their descendants share. To avoid race conditions, in case of descendants change the shared state, a *serialized super send*[De Meuter, 2004] construct has been included which allows expression to be executed atomically in the context of the shared parent.

ChitChat Synchronization Synchronization in ChitChat is summarized as follows [De Meuter, 2004]:

- Messages to passive objects are always sent synchronously.
- Messages to active objects are always sent asynchronously.
- Asynchronous messages return immediately a promise to the sender of that message.
- Delegation along a chain of passive objects always proceeds synchronously.
- Delegation along a chain of active objects proceeds asynchronously, which means, the method is immediately searched for but the activation is scheduled in the queue of the object that holds the method implementation.
- `athis` and `asuper` functions are created in a similar sense to `this` and `super`. The difference is that `athis` and `asuper` are handled asynchronously and return a promise. `this` and `super` refer to passive objects within the active object executing them. `athis` and `asuper` refer to active objects, which can be either local or remote.

4.6.2.2 Distribution in ChitChat

Chitchat uses the broadcasting approach to get an initial network reference to an object residing on a different machine. This approach is inherent of decentralized network architectures as P2P ones. As said in chapter 3, these decentralized schemes are more suitable for open networks than centralized name servers. ChitChat provides a discovery service mechanism in which active objects can register themselves to a channel by using `register('a Channel Name')`. It will produce a continued broadcasting of the active object in the channel. Active objects can discover their neighbors by calling `members('a Channel Name')`, that creates a local table containing the network references of the other active objects. Finally, they can disjoin the channel by invoking `unregister('a Channel Name')`. This form does not invalidate references already established to that object from within other machines.

Active and Passive Objects Through the Network These two types of objects follow different patterns when they are sent trough the network (as arguments or return values). This distinction is also related to the containment principle described in the previous section:

Passive objects are passed *by copy* This is assumed to insure active objects to be the main source for distribution and passive objects to remain encapsulated inside active ones.

Active objects are passed *by reference* It means that a unique reference to the active objects is passed through the network, instead to pass the active object itself. It remains at its original device.

4.6.2.3 Evaluation of ChitChat

The ChitChat programming language has been a strong influence for this work due to the close relation in goals between [De Meuter, 2004] and this thesis (the creation of a mobility model). De Meuter discusses different models of objects, concurrency and distribution found in programming languages, in the aim to provide code strong mobility in open networks. Such a work not only discusses but also presents its own proposal based on its research. Hence, ChitChat implements the classless object-oriented model of Pic%, the concurrency model based on active objects defined by ABCL, and the distribution model based on the implementations made by Emerald, Argus, and others.

ChitChat has the following properties:

- ChitChat makes a distinction between active and passive objects. While active objects are the entities with processing capacity that can interact over the network, the passive objects are used to implement (with traditional object-oriented modeling techniques) the behavior of the active objects.
- It also proposes a set of rules to work with these objects when creating systems for open networks. Active objects are always passed by reference. Instead, passive objects are passed by copy.
- It establishes the mechanisms for delegation between objects (both active and passive ones), adding new abstractions for supporting delegation of active objects over the network.

However, ChitChat has the following issues, regarding the requirements of open networks described in chapter 2:

1. Active object are neither aware to the resources of the devices that contain them (connection to the network, battery power level, etc.).

2. ChitChat does not provide any mechanism to discover other actors over the network.

4.6.3 SALSA

SALSA[Varela, 2001] (acronym for *Simple Actor Language, System and Architecture*) is an actor-oriented language based on Java created by Carlos A. Varela at University of Illinois in 2001. This language was aimed to be an small extension of Java that allows programmers to build internet and mobile computing applications using the advantages of the actor-based programming. The advantages are [Varela, 2001]:

- Actors are autonomous units with full encapsulation of its state and processing.

- Actors communicate asynchronously and do not share any memory.

- Actors provide a unit of concurrency by processing one message at time.

These properties are taken into account to implement mobility and application reconfigurability in SALSA (described in chapter 5).

4.6.3.1 Concurrency in SALSA

The primitives of the Actor model are supported in SALSA as following:

Actor Creation SALSA programs are grouped in modules which contain actor interfaces and behaviors. Behavior definitions can contain constructors used in actor initializations. Actors are constructed in a similar way as object are constructed in Java (using the `new` primitive). This actor creation statement is one of the three approaches to get an *actor reference* in SALSA, (concept useful for its distribution model explained in the next section).

Message Passing Message passing in SALSA is implemented by asynchronous message delivery with dynamic method invocation. This statement requires a target (an actor reference), a reserved keyword (`<-`), and a message handler (a SALSA method) with arguments to be sent. The following examples are two variants to message passing in SALSA:

```
    method(); // equivalent to "self <- method();"

    standardOutput <- println("Hello World")
    // standardOutput is an actor reference
```

Support for Actor State Modification *An actor changes its state by updating its internal variables through assignments or local method invocations*[Varela, 2001]. Only an actor itself can change its internal state, since all variables are private. Actors can change the states of other actors only by message passing.

Coordinating Concurrency In SALSA there are three types of continuations (which correspond to the concept of promise pipelining described in section 4.3.2.1) that are used for coordinating interactions between actors: *token-passing*, *join* and *first-class* continuations. The *token-passing continuation* corresponds to the continuation mechanism described in section 4.3.2.1. It is used in SALSA for specifying the order of processing. The value returned by a computation to its continuation is called `token` and is represented by the keyword `@`. Note that this simple scheme enables chains of continuations.

```
  checking<-getBalance() @ savings<-transfer(token);
```

A *Join continuations* joins tokens returned by multiple actors, once the have finished processing their messages, returning an array with these tokens to the continuation actor.

```
  join {
    standardOutput <- print("Hello ");
    standardOutput <- print("World");
  } @ standardOutput <- println(" SALSA");
```

Finally, a *First-class continuation* delegates a computation to a third party, enabling dynamic replacement or expansion of messages grouped by token-passing continuations.

```
//Example of using First-Class Continuation
  ...
  void saySomething() {
    standardOutput <- print("Hello ") @
    standardOutput <- print("World ") @
    currentContinuation;
  }
  ....
//statement 1 in some method.
  saySomething() @ standardOutput <- print("SALSA");
```

4.6.3.2 Distribution in SALSA

The distribution model in SALSA is known as *Worldwide Computing Model* [Varela, 2001] which enables actor programs to be open and dynamically reconfigurable. It involves universal naming, theaters, service actors, migration and concurrency control.

Worldwide Computing Model Worldwide Computing (*WWC*) Model is a global distributed infrastructure enabling naming, communication and migration for actors. This is a actor-based model that consists of a set of virtual machines (known as *theaters*) hosting one or more actors (called *universal actors*) and providing a layer for message passing and remote communication.

Universal Naming Universal naming is a strategy to make actors reachable on a network. This service is in charge of providing object name uniqueness, allocation, resolution and location transparency. There are three main components in this naming system: universal actors, universal naming service protocol and universal actor references.

Universal Actors This mechanism allows actors to become *universal actors* [Varela and Agha, 2001] by providing them with a unique and location-independent name known as *Universal Actor Name* (acronym UAN), which is mapped to the actor's current locator called *Universal Actor Locator* (acronym UAL), a uniform and unique handler containing location and protocol information for communication with the given actor.

A sample UAN for an actor handling a printer:
`uan://osl.cs.uiuc.edu/˜agha/printer/`

A sample UAL for this actor:
`rmsp://agha.cs.uiuc.edu/myPrinter`

This naming scheme was motivated by the scalability and readability afforded by the world-wide-web's addressing approach for uniformly identifying multiple resources using Uniform Resource Identifiers[5] (URI) proposed in [Varela, 2001].

Due to independence between Actor's UAN and UAL, when it migrates from one theater to another, its UAN remains, but its UAL is updated in the corresponding naming server to reflect the new location (see figure 4.4). The migration is transparent for other actors that had gotten already a reference to the migrated actor (see more details about actor migration in SALSA in the next chapter).

[5]URIs includes Uniform Resource Locators (URL) and Uniform Resource Citations (URC) containing metadata. Nowadays, only URLs are widely used.

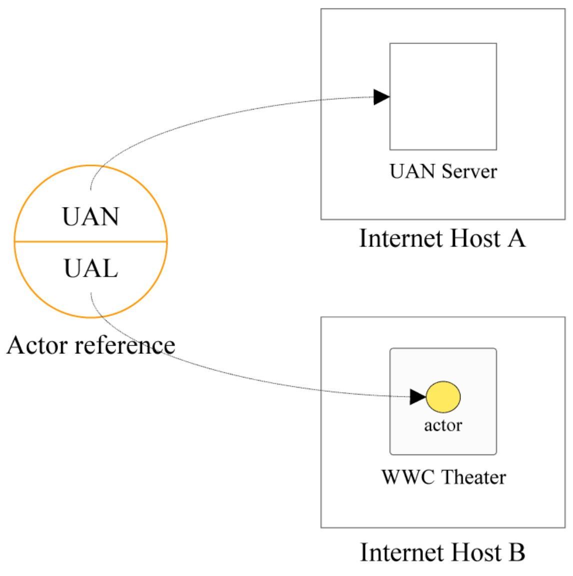


Figure 4.4: UAN and UAL independence

Universal Naming Service Protocol Universal Naming scheme offers a naming service and protocol to implement the mapping from names (UANs) to locators (UALs). Initially this service was implemented in a centralized way, by using a set of servers which understood the *Universal Actor Naming Protocol* (acronym UANP). Later works in SALSA have included distributed naming strategies such as the implementation of a *fault-tolerant home-based naming service*[Tolman, 2003] based on Chord P2P-lookup protocol presented in chapter 3 (note that Chord was precisely designed for efficiently working with mobility).

UNAP provides basic methods for creating, accessing, updating and removing entries in a (distributed) name server. These entries are pairs composed by the UANs and UALs of the universal actor. This protocol is not used directly but through a high-level programming language abstractions described in the following point.

Universal Actor References Universal naming service provides high-level abstractions for enabling the association of actor references in SALSA language to the programming-language-independent actor names (UANs) and locators (UALs). These universal actor references make the actors to be accessible to other universal actors. The high-level abstractions are methods included in the behavior of universal actors which support the following primitives:

- Binding an actor to a name and an initial theater.
- Getting references to and communicating with a universal actor
- Migrating an actor from one theater to another.

SALSA Theaters The virtual machine where universal actors reside is called *theater* (see figure 4.5). It enables the execution of universal actors. A theater contains the following components:

- A *Remote Message Sending Protocol* (RMSP) server for remote communication and messenger migration (described at the end in this section).
- A runtime system for universal and environment actors. This last type of actors corresponds to those that enable the access to the immobile resources of a theater, such as standard output and input streams.

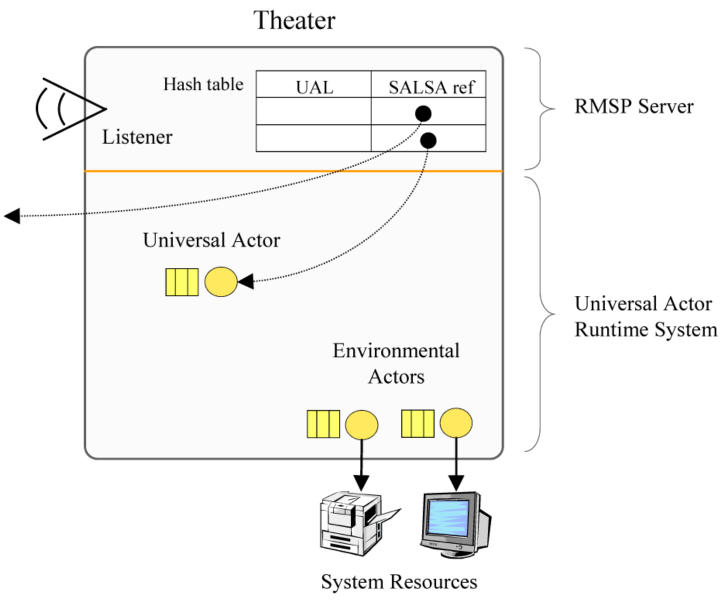


Figure 4.5: SALSA Theater

In the newer implementations of the universal naming service[Tolman, 2003] based on a P2P architecture, this service is incorporated into the theaters themselves (instead of using dedicated naming servers). To achieve this a *stage manager* is included in each theater who, upon receiving a naming request, delegates responsibility to serve the request to a *stage hand* that is a chosen from a thread pool.

Remote Communication Universal actors communicate between them by using the *Remote Message Sending Protocol* (RMSP). It is an internet-based protocol implemented as an extension to Java object serialization. Each theater contains a RMSP server that listens for incoming messages from abroad for actors living on this theater. It keeps track of hosted actors and their locators (by using a hashtable with the UAL as key and the SALSA reference to the local actor as value), so that incoming messages can be properly passed to the target actor. Note that there are no guarantees that ensure the target actor of a message will not move in the mean time the message left the sender actor's theater and is traveling over the network. RMSP provides a solution to this situation, which will be explained in the context of universal actor mobility in the next chapter.

4.6.3.3 Evaluation of SALSA

SALSA is an actor-based language that has adapted the actor model to work with internet applications. Its main contribution is the distribution model (WWC model) that provides a set of abstractions for dealing with open networks:

- It uses the universal naming strategy to make actors reachable on the network.
- Universal actors are provided with a location-independent name (UAN) which is mapped to a uniform and unique handler (UAL) containing information about its location.
- This language provides a protocol (UNAP) to discover universal actors based in the universal naming strategy. Actually this service is implemented using a DHT-based P2P scheme.
- The SALSA theater is an abstraction to the virtual machine where an actor resides. It provides mechanisms to both interact with resources found at this location and communicate its resident actors with others over the network.

Respect to the applicability of SALSA in mobile open networks, it bases its strategy in the implementation of the DHT-based P2P architecture. As explained in chapter 3, this is a proper architecture for maintain a consistent state of systems running over open networks, considering its dynamic reconfigurability. But this conditions is also related to the dynamic configuration of the services found (offered by actors) in the network along the time. SALSA does not provide mechanism to discover and interact with these services. The following case presented in this chapter will introduce explicit mechanism to deal with service discovery and actor communications over an open network.

4.6.4 AmbientTalk

AmbientTalk[Dedecker, 2005a] is an actor-based programming language for *ambient aware programs* created by Jessie Dedecker at VUB in 2005. It extends the actor model by including characteristics of the ambient-oriented programming paradigm. The idea behind this language is to build programs that *can sense the environment (ambient) and interact with it* [Dedecker, 2005a].

Since it is the language on which the work of this thesis has been created, this chapter will dedicate more attention to this language, as well as the paradigm included in its actor model extension.

4.7 Ambient-Oriented Programming

Dedecker defines on ambient-oriented programming (acronym *AmOP*) language as a concurrent distributed object-oriented programming language that is specifically geared towards the use of mobile devices. To accomplish this it must employ *Non-blocking Communication Primitives*, *Reified Communication Traces* and the *Ambient Acquaintance Management*. All of these properties have been implemented in AmbientTalk language, which is described in the section 4.7.1.1.

Non-Blocking Communication Primitives As explained in section 4.4, blocking communications may produce distributed deadlocks in open networks where participants such as mobile devices are not necessarily available during the communication process (this phenomenon known as connection volatility was explained in chapter 2). As remarked in [Dedecker, 2005a], *having blocking communications primitives would imply a program or device to block upon encountering unstable connections or temporary unavailability of another device*. For this reason, an ambient-oriented language include in its concurrency model non-blocking communication primitives which minimizes the time resources are locked.

A non-blocking mechanism is not only asynchronous communication system, since the later provides a *send* non-blocking operation, but says nothing explicitly about a *receive* operation of the message in the target actor. The tuple-space based middleware (described in chapter 2) shows a case of the combination between a non-blocking *send* and a blocking *receive* operation.

Reified Communication Traces The communication process is decomposed in parts representing the different status of a communicated message. Non-blocking communication between autonomous devices (see more details about device autonomy phenomenon in chapter 2) might imply the communication between two parties ends up in a state that is not consistent for the purposes the communication was started (for instance when a message is received but not processed by the target actor because it was shut down before the message was served). *Whenever such a inconsistency is detected, the parties must be able to restore their state to whatever previous consistent state they were in, such that they can decide what to do based on the final consistent state they agreed upon*[Dedecker, 2005a].

Reified communication traces in a ambient-oriented programming language provides a *explicit representation* of the communication state allowing to react to inconsistencies detected during this process (for instance, reversing part of its computation).

Note that this mechanism does not give full guaranties that a communication will be successful. There have been large studies that concluded it is not possible to insure the correct delivery of a message [Dedecker, 2005a]. The reified communication traces just offer programmers a way to manipulate the state of an actor communication in order to achieve application-specific recovery.

Ambient Acquaintance Management This property is related to the fact that resources are dynamically detected as devices find each others. Each device is said to have capabilities to interact with others. These capabilities can be understood as services that can be required by other devices. Interactions between devices are direct ones. There is no need to rely in third parties to get acquaintance of other actors (like in the cases of chat-servers or white-boards client-server applications). It implies to have a distributed naming protocol like in P2P architectures[6].

4.7.1 AmbientTalk Kernel

AmbientTalk is ambient-oriented programming language that implements the ambient actor model explained in the previous section, which is an extension of the actor model. As mentioned in section 4.7, an ambient-oriented language is also a concurrent distributed object-oriented language. As such, the implementation of AmbientTalk considers the following parts:

- The implementation of the object model, which is based on the object model of the programming language Pic%[De Meuter et al., 2004] (like ChitChat). This model is based on the prototype-based object paradigm. Comparisons between class-based and prototype-based approaches have concluded that the latter have more advantages for building applications in open networks [De Meuter, 2004].

- The implementation of the actor model. AmbientTalk provides three primitives that implement the three basic operations found in the original actor model:

    1. A new actor is created with the `actor` primitive which takes an object as one argument and returns a reference to the new actor with the object as its behavior.
    2. Messages are sent by using the # keyword which represents the asynchronous method invocation operator.

[6]It is still possible for programmers to set up servers for ambient-oriented applications. This property is related only to the fact that acquaintances of an object must be dynamically manageable[Dedecker, 2005a].

3. Finally the state and behavior of an actor is changed by using the `become` primitive. It takes on object as argument that will be in charge of processing the following messages.

- The implementation of the ambient actor model to accomplish with the properties of an ambient-oriented programming language described in the previous section. The definition of this new model is presented in the next section.
- The implementation of a set of functions to enable reflection in messages and mailboxes, at communication and processing stages.

4.7.1.1 The Ambient Actor Model

As mentioned at the beginning of this chapter, the Ambient Actor model (acronym AAM) is an extension to the Agha's actor model. AAM extends this model by including mechanism to accomplish with the properties of the ambient-oriented paradigm described above. In [Dedecker, 2005a], Agha's actor model is evaluated respect to these properties. It behaves as follows:

- The actor model partially fulfill the property of non-blocking communication primitives. This is because, as explained in section 4.3.2, this model only have support for the *send* operation but it says nothing about the *receive* operation.
- The actor model does not support reified communication traces, because it does not identifies states in the actor communication process.
- The actor model requires third-party actors to gain new acquaintances of the network. A later extension to this model also developed by Agha, called the *ActorSpace model*[Callsen and Agha, 1994], enabled distributed naming by introducing an actor grouping mechanism, called *spaces*. However these spaces are still managed by centralized authorities.

The ambient actor model accomplishes these properties by incorporating new mailboxes to the model which allow to make the communication state of an actor explicit, and for ambient acquaintance management.

Communication State Ambient actor model enables control over the communication state of an actor. According to this state a message can be in one of the following status of delivery:

- A message was received by a target actor but it (the message) is still not processed by this actor. This message is actually stored in a mailbox called `in` which corresponds to the queue found in the original actor model.
- A message was received and processed by the target actor. This message is stored in a mailbox called `rcv` (acronym for *received*).

- A message was sent by an actor but it is still not transmitted to the target actor. This message is stored in a mailbox called `out`.
- A message was sent and transmitted from the sender actor to the target actor. This message is stored in a mailbox called `sent`.

This mailboxes are a kind of *gates* to the past and future of the current status of the actor.

Ambient Acquaintance Management Distributed naming is available in AmbientTalk via a P2P lookup mechanism called *pattern-based lookup*. A pattern is an abstract description of an actor (or a set of them), and is specified by a communicable value. Note that usually in P2P lookup mechanisms this description is the name of the actor or a key understood by a P2P lookup algorithm (such as DHT-based algorithms explained in chapter 3). However, this description could perfectly correspond to a service offered by an actor which, in some scenarios, can make much more sense than looking for its name. When somebody looks for a printer, it would be pretty more desirable for him to look for the *print* service over the network, than looking for *the name of the computer that has connected a printer* or *the name of the printer itself*.

The pattern-based lookup works as follow:

1. An actor that wants to search for certain pattern (for instance, representing an actor or an actor's service), put this pattern in a mailbox called `required`. It means, this actor *requires* such pattern.
2. An actor that wants to make some pattern available (for instance, representing itself or some service it provides), put this pattern in a mailbox called `provided`. It means, this actor *provides* such pattern.
3. When the actor providing a pattern enter to the communication range of another requiring the same pattern, this last actor joins to the provider actor. It is done by registering the tuple [”pattern found”, ”actor providing this pattern”] in other mailbox called `joined`. This tuple is called *resolution* in AmbientTalk.
4. Whenever an actor, that is already included in a resolution of another, is pulled out of communication range, this resolution is moved to another mailbox (of the same actor) called `disjoined`. This way, ambient acquaintance mechanism not only detect new resources, but also to detect when actors have disappeared from the network.

4.7.2 Evaluation of AmbientTalk

AmbientTalk is an ambient-oriented programming language for mobile open networks. It is based on the ambient actor model which provides the two following mechanisms:

- A mechanism to control the communication state of an actor. It provides the actor with non-blocking communications(for avoiding distributed deadlocks) and reified communication traces (for avoiding inconsistences during communication process due to the connection volatility of the mobile devices).

- A pattern-based P2P lookup mechanism for the discovery of the services provided by actors over the network (ambient acquaintance management).

These mechanisms have been included in AmbientTalk through the implementation of extra mailboxes (to the ones presented in previous actor model implementations) and reflective methods to operate over them. However, the programmer does not have necessarily to interact with these mailboxes directly. As will be explained in chapter 6 AmbientTalk allows programmers to build higher-level abstractions on top of this implementation.

AmbientTalk is a language that provides the mechanisms mentioned above to comply with the requirements of device mobility over a network. But it does not consider any implementation related to actor (or any other computation) mobility. Previous languages described in this chapter (sections 4.6.2 and 4.6.3) provide actor mobility as a way to deal with open networks. As explained chapter 6, the proposal of this thesis points to develop an actor mobility mechanism for the AmbientTalk language.

4.8 Conclusion

This chapter has described several concurrency and distribution models as well as their implementations in different programming languages and other frameworks. Traditionally there have been two different concurrency approaches: the thread-based and actor-based models. The latter has advantages over the former for dealing with open networks such as asynchronous communications less vulnerable to the connection volatility of devices, to the expressiveness, transparency of concurrency and distribution implementation details. However, both models still have issues for working with open networks. These models do not say anything about the way in which the dynamic reconfigurability of these networks produces an impact on the resources present in the network along the time. They neither deal with consistency of the communications upon device failures.

The ambient-oriented programming paradigm implemented by AmbientTalk solves the issues mentioned above by considering two mechanisms: communication states and the pattern-lookup method. Both are used in the AmbientTalk mobility model presented in this work (chapter 6).

Different programming languages and frameworks have adapted these models to the conditions of different types of networks. Adaptations of the actor model for open networks (MicroTAPAS, ChitChat, SALSA and AmbientTalk) have identified the following new elements:

- Distinction of the entities that can interact with others and move over the network (active objects, universal actors and ambient actors after this work), respect to the ones that are only used locally (passive objects, objects and environmental actors).

- Abstraction for the devices and their resources (theaters, capabilities, environmental actors).

- Actor Discovery Services (universal naming service protocol, director management, pattern-lookup mechanism).

The following chapter will present the mobility model of these languages.

Chapter 5

Strong Mobility in Open P2P Networks

5.1 Introduction

This chapter describes the reasons and requirements for code mobility in open P2P networks, and its implementation in different programming languages. Most of these languages were presented in the previous chapter in the context of the evaluation of their concurrency and distribution models. The analysis of their mobility models is relevant for the development of the model exposed in this dissertation. It is strongly based on the properties identified in such previous mobility implementations.

Code mobility is not new. In the late 1970s code mobility was already required for process migration [Milojicic et al., 2000]. In such a case, a process (operating system abstraction that encompasses the code, data and operating system associated with an instance of a running application) was transferred from one computer to another. Code mobility in that case was used to enable load distribution and fault resilience. Years later, mobility was related to agents which, as explained in the previous chapter, are programs that can move over the network and autonomously execute task on behalf of users.

This chapter first describes the reasons and requirements for code mobility in open P2P networks. After that, it defines different types of mobility. Strong mobility is one of them. It is explained in this chapter why this type of mobility is more adequate for this open P2P networks (particularly for mobile open P2P networks that are the scenario targeted in this work, described in chapter 2). The final part presents different programming languages that implement strong mobility of either objects or actors.

5.2 Code Mobility in Open P2P Networks

This section presents the code mobility in the context of the mobile open P2P networks. In the first part identifies some reasons for code mobility in these networks, comparing them with historical reasons for this mechanism. The second part reviews the properties of mobile networks explained in chapter 2 and discusses their implications for code mobility.

5.2.1 Reasons for Code Mobility in Open P2P Networks

Historically, code mobility has been used either to move processes, agents or any other computation towards desired resources or away from scarcity [Milojicic et al., 2000]. For process migration it was related to move a process towards a under-loaded computer or a specific resource (such as a database). In the case of agents, they could move toward a source of information for improving the performance (by accessing locally to data or other resources).

This reason is still valid for open networks. Code mobility can be useful for dealing with their dynamic reconfiguration due to the volatile connection of their participants. In mobile open networks, the connection volatility and resources scarcity of mobile devices is a motive relevant enough to move applications residing inside of them to other devices over the network with better conditions. In such a case, code mobility helps to ensure the availability of the services provided by mobile devices. It is even more suitable when working on P2P architectures because no servers are required to take the decision of moving computations and to perform the movement itself.

There is another reason found in the context of AmI [Lindwer et al., 2003] vision, mentioned in the introduction. Mobile and embedded technology will allow the users to stay connected to their services even when the users move around. It will imply to have applications hoping from one device to another in order to ensure such an availability. The chapter 7 will explain more in detail the AmI vision and will develop an example of this type of application that follows its user.

5.2.2 Requirements for Code Mobility in Open P2P Networks

Some properties of open networks described in chapter 2 need to be considered in the implementation of code mobility, in the following way:

- In a mobile open network the code mobility will coexist with the device mobility, it means that during the move process of a code (like an application) there could happen two unexpected changes of conditions in the network:

    1. The device where the application is going to move get disconnected (because its connection volatility or scarce resources).
    2. The device where the application resides get disconnected before the application moves completely.

- Devices in open networks are heterogeneous. As such these devices can vary in their resources (such as screen definition battery power, etc.). The move process of an application should consider this difference by implementing some kind of reconfiguration and rebinding mechanism right after the arrival of the application to a new device.

5.3 Types of Code Mobility

There is a criteria for classifying the different types of code mobility described by Fugetta et al. in [Fugetta et al.,]: it is related to the computational context of running programs which is the knowledge a language processor has about the program it is executing. This context considers environments, working memory, runtime stack and so on. Fugetta et al. identify three types of contexts:

Data context It is known as the state of a program, which is composed of the set of variable bindings that are accessible and allocated for the program at some requested moment.

Control context It is related to the status of the computation. It is composed of a reference to a *point in the code* (like a program counter or a current expression), and possibly a description of those *past states* of the computation that are still relevant (typically a runtime stack or a continuation).

Resources context It includes all the bindings of a program in memory that are not allocated for the program alone. Those bindings can be in the internal memory (like operating system resources), or can be external resources (such as databases).

Based on the amount of contextual information that is transmitted upon migration it is possible to classify mobility mechanisms. Fugetta et al. summarize these mechanisms by distinguishing four kinds of computation mobility: Weak mobility, semi-strong mobility, strong mobility and full mobility. The following sections describe and compare them; the table 5.1 shows the relationship between these different types of mobility and the computational contexts.

5.3.1 Weak Mobility

This kind of mobility does not include any computational context, it merely sends *dead code* over the network. When code arrives at its destination machine, it starts running as if it never ran before.

Weak mobility is exemplified by Java applets, which are chunks of stateless code downloaded from a web sever and executed on the client machine. Since only the code is transmitted the applets do not have any state from previous runs (except for some static bindings like constants).

	Computational Context		
	Data Context	Control Context	Resources Context
Weak Mobility	∅	∅	∅
Semi-Strong Mobility	✓	∅	∅
Strong Mobility	✓	✓	∅
Full Mobility	✓	✓	✓

Figure 5.1: Computational Context

5.3.2 Semi-Strong Mobility

This is the type of mobility used by most of middleware solutions (described in chapter 2). It considers the data context in the mobility process, allowing computations to move from one device to another by using special mechanisms for halting and resuming the moved computation, right before and after the movement. This means that semi-strong mobility does not support control context, which must be *manually copied* into the data context and upon arrival of the computation (with its data context) at the new location, making sure this control context will be correctly restored.

Encoding the control context in the data context can be complex, since it implies to have algorithms explicitly divided to allow for mobility. In order to move a computation, programmers not only have to serialize it at different points of its data context (with the control context already encoded inside), but they also have to encode the *future* of that computation once it arrives to the new location (known as continuation passing style programming described in chapter 4).

Another example of semi-strong mobility is the Java serialization API. In Java an object graph can be serialized, but the control state of java.lang.Thread is not copied.

5.3.3 Strong Mobility

This kind of mobility includes the data and control contexts of a computation. It allows running process to hop from one machine to another without manually halting the computation it is performing.

Strong mobility is associated to *autonomous agents that roam networks to accomplish a certain task on behalf of their owner* [De Meuter, 2004]. Section 5.4 gives some reasons to state that this type of mobility is also suitable for open networks, and the computation to be moved can perfectly be autonomous actors which, as explained previously (chapter 4), are much simpler entities than agents.

5.3.4 Full Mobility

This kind of mobility considers the whole computational context of a computation (data, control and resource contexts). Programs in full mobility approaches are said to be moved with the complete closure of its data and resources contexts without any interruption in its current execution.

In full mobility resources are never rebound, which according to [De Meuter, 2004] can be done by *moving them*, *making a copy*, or *creating a reference to them from within the new machine*. It means that moved computations will not require any interaction with the new machine, becoming a kind of *parasite* computations that only uses the machine as a source for computational power. This technique is useful in the context of process migration for load balancing purposes, but it does not seem to be suitable for open networks in which rebinding mechanisms and bindings of new resources are opportunities to keep the services required by users available.

5.4 Reasons for Strong Mobility in Open P2P Networks

Complementarily to the reasons for code mobility in open P2P networks described in 5.2.1, there are other reasons that justify the use of strong mobility in these networks. The reasons are the following:

Intentional strong mobility It is related to the explicit necessity of considering mobility aspect as part of a solution for the design of a program. It can be required for either *resource optimization, software design considerations* or *identity preservation* [De Meuter, 2004].

As a way to optimize resources, strong mobility is used as a load balancing mechanism which, unlike full mobility, can rebind some required resources upon arrival at its new destination.

Design considerations can lead designers to move objects along with others that *logically belong together*. Strong mobility is desirable in this case since these objects are currently active (they are on some runtime stack already).

Finally, the identity preservation is related to the scenario in which objects are moved by reference (for instance as arguments of a remote method invocation), that means no copy of them are created. All references to the object at the sending machine must refer to it before and after its movement.

Strong mobility for actors Working with actors in distributed systems unavoidably implies to deal with actor mobility. A natural scenario for this context is having actors that are passed as arguments in remote method invocations (messages). These actors may have to be strongly moved since they are running computations whose queue might be filled with request. It has been the reason why mobility has been included in actor-based programming languages.

One good reason for moving an actor when passing it as a parameter is when this actor will be sent lots of messages on the remote machine. If the actor is moved, this will reduce the amount of remote method invocations back to the sender machine because the actor will be local to the receiver machine.

Any method in an actor-based system can demand actors to be moved. It can be a source of problems if actors do not carry with themselves their control contexts in order to continue their executions. This situation gets even worse if it is considered that the method calling an actor to move does not belong to this actor, as described in [De Meuter, 2004].

5.5 Strong Mobility in Object-Oriented Languages

The following cases are object-oriented programming languages that have implemented strong mobility. These are Telescript [White, 1996], Obliq [Cardelli, 1995] and Emerald [Jul et al., 1988].

5.5.1 Telescript

Telescript [White, 1996] created at General Magic is a class-based programming language created at in. This language works with classes which can be declared `abstract` or `sealed` (like `final` classes in Java).

Security is the only particularity of Telescript with respect to standard class-based languages. It is achieved by using four built-in *mixins* (named abstract subclasses that can be applied to several classes in row), which can be applied to classes to modify them:

- `unmoved` mixin renders objects of the modified class unmovable.
- `uncopied` mixin turns a class into a class whose instance cannot be copied.
- `copyright` mixin turns a class into one that can only be instantiated by *copyright enforcer* [De Meuter, 2004].

- `protected` mixin turns a class into one whose instances cannot be modified.

5.5.1.1 Mobility Model in Telescript

Telescript features strong mobility by providing a class hierarchy composed by the superclass `Process` and its subclasses `Place` and `Agent`. *Places* host *Agents* which have a method `go` to move. It contains a `Ticket` object as parameter. This parameter determines the trip of the object upon invocation of `go`.

There is an alternative method called `send` used to move copies of an agent to different locations. It looks like the mobility programming pattern master-slave in which an object, that is charged with a certain task, can spawn a number of mobile *envoys*. In parallel, these mobile *envoys* accomplish the task partially at another location.

Telescripts implement a security model based on capabilities which are stored in a object contained in the `permit` attribute of an `Agent`. An agent's `permit` defines a set of predefined boolean values such as `canCreate` (defining if an agent can create new process), `canGo` (if an actor can move around), `canGrant` and `canDeny` (if it can raise or lower the permision level of the other processes).

5.5.1.2 Evaluation of Telescript

Telescript identifies in its mobility model two entities: places and agents. Agent mobility can be achieved by using two methods: `go` and `send`. One of the interesting features of Telescript is its security model based on capabilities. These capabilities can be useful for reconfiguration purposes of agents (or any other computations) after their arrivals to new locations. A similar capabilities-based security mechanism could ensure that agents will have restricted possibilities of altering the conditions at the new location.

5.5.2 Obliq

Obliq [Cardelli, 1995] developed at DEC is a lexically-scoped and untyped language developed by at in, for writing computations that can roam networks. Obliq objects are records of named slots of the form { `x => ..., m => ..., ...`} where each slot is either a value field, a method invocation or an alias to a slot of another object. This language is prototype-based but does not feature delegation or object-based inheritance. Objects in Obliq are created by adding attributes to copies of existing objects (by using the operator `clone` as described below).

There are four language operations in Obliq that can be applied to local or remote objects:

Selection and Invocation This operation has two variants according to what is found in the slot.

`a.x` selects a value from the field `x` and return it.

`a.x(`b_1`, ... ,`b_n`)` invokes a method from the field `x` of `a`, supplying parameters and returning a result.

Updating and Overriding This operation deals with both field update and method override:

`a.x:=b`.

In this example `x` is either a field or a method of `a` replaced by `b` and `b` is either the new value or method.

Cloning It clones an object and possibly adds new slots to the clone.

`clone(`a_1`, ... ,`a_n`)`

Delegation or aliasing It is the operation that replaces fields with aliases.

`{x => alias y of b end, ...}`

Obliq locates objects using a central name server in which objects are registered through the use of the operation `net_export(''objectName'',DNS,object)`, and located with the operation `net_import(''aCertainName'',DNS)`

5.5.2.1 Mobility Model in Obliq

Obliq mobility is accomplished by combining the cloning with aliasing operators. A self-synchronized object that wants to move:

1. It sends to a remote execution engine residing on the receiving host, a first class procedure that contains as its body a `clone` operation, such that the clone is constructed at the site of the execution engine. The value returned by this remote `clone` operation is not the clone of the object itself but a reference to it.

2. Obliq takes a reference to the cloned object for both sending it as result of the cloning operation and for insuring that the clone is constructed locally.

3. After the returning from the execution of the cloning operation, the object that spawned the procedure has to redirect all its fields to refer to the remote clone. It can be done by using one atomic instruction that is syntatic sugar for several aliasing instructions.

Part of the trick here is that Obliq's objects are required to move themselves because no thread should be able to tinker with the object during the execution of these steps. It therefore all has to happen atomically. Obliq enables this by declaring the object as self-organized so that the method that causes the copying and redirecting cannot be interrupted. [De Meuter, 2004].

5.5.2.2 Evaluation of Obliq

Obliq is a distributed object-oriented programming language that provides mobility by combining cloning with aliasing higher-order functions. It allows to create remotely objects and get back a reference to it. Obliq uses a centralized name server where objects are registered (like Java RMI) that as explained in chapter 3 is less convenient for open networks than P2P schemes. However, there is nothing in the uses of the central server that cannot be transformed into a decentralized naming server (SALSA was initially used a central naming server too and now it uses a DHT-based P2P scheme, see section 5.6.3).

5.5.3 Emerald

Emerald [Briot and de Ratuld, 1988] is a distributed programming language developed at University of Copenhagen in 1988. It was mainly designed to support applications that had to run on a fixed and well-defined network topology. Although this language is a prototype-based language, it provides classes, but the keywords are mere syntactic sugar for the creation of a factory prototype (representing the class) and a description of the types of its *instances*. At the same time, Emerald is statically typed. This is combinable with prototypes because of the absence of dynamic features like delegation.

Each object in Emerald contains a unique network-wide name, a representation containing the data local to the object (primitives and references to other objects), a set of operations that can be invoked on the object, and an optional process, which converts a object in an active-object (by starting a thread of control).

Emerald's concurrency model is a combination of active objects (or actors) and threads that enables the following actions:

- An object can implement *process blocks* which contain code that will be entirely executed autonomously by a independent process.
- An object can be declared *monitor* guaranteing mutual exclusion between their operations.
- Explicit synchronization is achieved by using the `wait` and `signal` operations in a similar way the `wait` and `notify` methods are used in Java.

Emerald support partial failures by including a `checkpoint` statement that saves the state of an object to stable storage to facilitate recovery after a potential crash. It also provide *failure handlers* in the same sprit of try-catch constructs found in Java or C++.

Emerald kernel ensures the location transparent message passing by assuming the responsibility of locating the receiver and handling the message properly after reception. Providing objects in the system with a unique name is useful for achieving this distribution transparency.

5.5.3.1 Mobility Model in Emerald

Emerald provides a small set of primitives that combined can produce different flavors of mobility. These primitives are `locate`, `move`, `fix`, `unfix` and `refix`.

- `locate x` returns the location (a `node` object) of the object x.
- `move x to y` co-locates object `x` with the `node` correspondent to `y` (which is not necessarily a node itself).
- `fix x at y` fixes object `x` at node correspondent to `y`.
- `unfix x` makes `x` mobile again.
- `refix x at z` unfixes automatically the object `x` from the current location and fixes it in the `node` correspondent to `z`.

The `move` primitive is just a *hint* in Emerald, which means *the kernel is not obliged to perform the move and the object is not obliged to remain at the destination site* [Jul et al., 1988]. It takes an object and passes it *by move* instead of *by reference* or *by copy*.

Arguments in invocations of Emerald methods can be annotated with the keywords `move` or `visit` that will imply that upon returning from the method call, only arguments annotated by `visit` will come back to the node of the sender of the message.

Emerald objects can be attached to other objects such that they move together. This is achieved by means of the `attached` modifier which can be used to annotate a variable declaration in the same spirit Java allows one to annotate variable declaration with a `static` modifier. An *attach* is a unidirectional relation, that is, if an object `o1` is attached to an object `o2`, then when `o2` moves, `o1` will follow. However, if `o1` moves, `o2` will not necessarily follow, unless it was also explicitly attached to `o1`.

5.5.3.2 Evaluation of Emerald

The main characteristic of Emerald programming language is its `move` primitive as a hint, it means that the move could not occur and the moved objects may return to its old location. The drawbacks of this language are the use of a unique name within the network, that can lead to problems when unforeseen devices containing objects with the same name enter the network. Another inconvenient for its use in open networks is its synchronous communication model, considered harmful for the autonomy of the devices (see chapter 4).

5.6 Actor-based Languages for Mobility in Open P2P Networks

The following actor-based approaches were already described in the previous chapter. The following sections will explain their mobility models.

5.6.1 MicroTAPAS Library

For MicroTAPAS mobility is *the most important feature needed to achieve adaptability and flexibility in the execution of service components*. Its mobility architecture (based in TAPAS [Shiaa and Aagesen, 2002]) considers the three following types of mobility: personal, terminal and actor mobility. For supporting them this architecture prepares some actors with specific roles (behaviors) in the coordination of the participants of a network.

5.6.1.1 Personal mobility

Personal mobility involves users that use some subscribed services from different location in the network. The goal of this type of mobility is to provide support to users for suspending and resuming their so called *sessions* [Luhr, 2004] from any point of the network without dealing with technical stuffs such as restrictions in private networks, security problems to execute their personal transactions and so on. Although the systems proposed by MicroTAPAS for personal mobility are based on actors, they do not rely on actor mobility and they are therefore outside the scope of this work.

5.6.1.2 Terminal mobility

Terminals (or mobile devices) in TAPAS are considered to be the interfaces towards the end user through which they can access their services while on the move. These services can consist of a group of actors working together, distributed over different nodes in the network. Hence a terminal executes an actor called *Mobility Agent* (acronym MA), that is responsible for tracking the terminal location, and another known as *Mobility Manager* (acronym MM) which receives the communications of the *Mobility Agent* and keeps updated all the references of the nodes participating in the services accessed by the user from its mobile terminal. A MA can send two types of message to its MM:

- `locationUpdate` message is sent upon changing terminal location.
- `nodeDiscovery` message is sent once a communication is required with other terminal.

Distribution of mobility entities Figure 5.2 shows the distribution of MA and MM in a wireless network. Unlike the original TAPAS architecture which relied on directors residing on servers to manage mobility, the introduction of wireless network gives rise to the following model:

1. MicroTAPAS includes the mobility entities into its model. Each node that is not a server must have a MA. This way, each node will be able to notify changes in its location.

2. A MM must be added at the same node in which the director of the domain resides (the server). Thus, each domain will have at least one MM providing mobility support.

3. MicroTAPAS includes the concept of *sub-domain* which deals with the dynamic configuration of wireless networks. The idea is to breakdown the pure client-server architecture present in TAPAS, converting it in a kind of hybrid P2P architecture (explained in chapter 3) in which *supernodes* should contain one MM providing mobility support to its correspondent *sub-domain*. This way, the *director* will continue to work in a client-server style, while MAs will use this hybrid P2P structure to offer their mobility services, and in this way overloads will be avoided in case of big systems.

5.6.1.3 Actor mobility

It stands for the movement of instantiated functionality at a node along its properties (such as behavior, capabilities, role-sessions. See chapter 4) that are executed by the actors living on it. The reasons in MicroTAPAS for actors to move around theaters, are basically related to reacting to the changes in those properties. Some of these changes can be: changes in the capability requirements, deterioration in the resource availability, dynamic change in the network configuration, or implications of terminal mobility. The mobility entities (MA and MM) will play an active role in actor mobility too.

Extended Actor As mentioned in chapter 4, MicroTAPAS extended the traditional actor model to include new components like capabilities and role-sessions. In addition, every component is classified according to the way it is moved to the new location. The two groups of actor components identified by this classification criteria are:

Moved (copied) components The components that will be copied to the new location of the actor. These are: state, capabilities and role-sessions. The queue of messages is dismissed in the current version of MicroTAPAS, which implies that it does not implement strong mobility of actors.

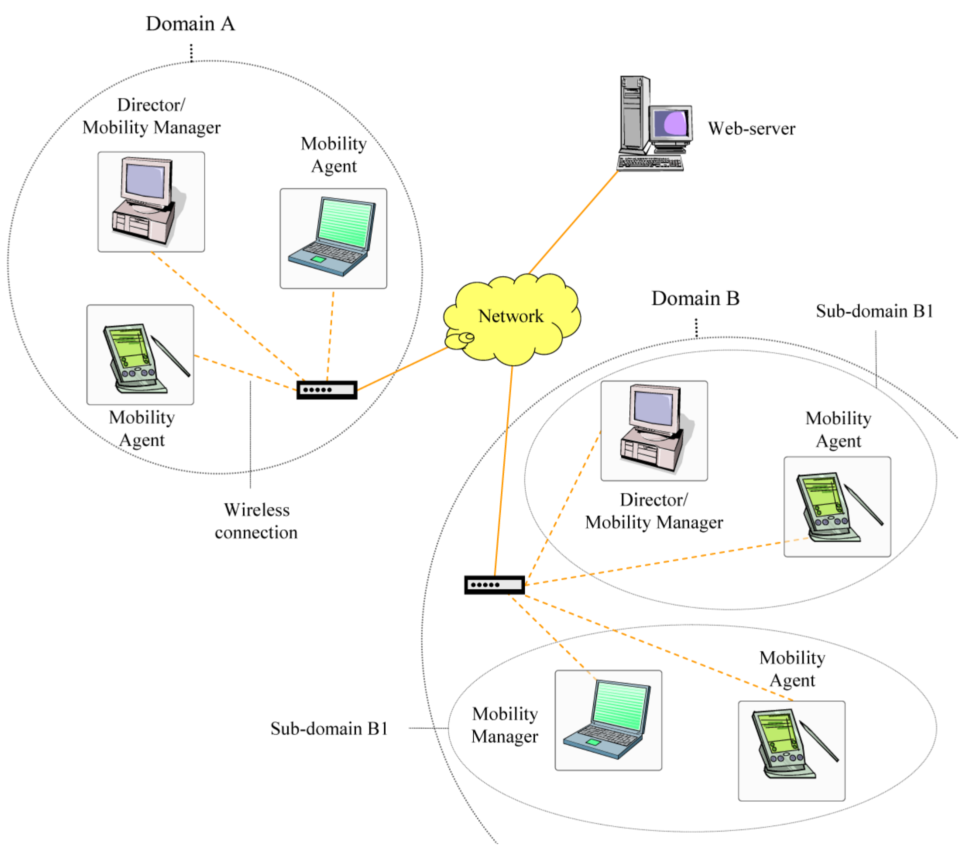


Figure 5.2: Distribution of mobility entities

Re-instantiated components The components that will not be moved to the new location of the actor and that will be subsequently re-instantiated. These are: methods and behavior. Note that MicroTAPAS makes a distinction between the implementation of the methods and their call definitions (called the behavior)

The move procedure The actor mobility process in MicroTAPAS is closely related to the components mentioned above and the ways to be moved.

1. As in terminal mobility, every communication must be preceded by an `actorDiscovery(actor1)` method.
2. An actor receives a `actorMove(location2)` message.
3. `capablityChange(capabilities)` allows to recover the capabilities (e.g. printers or screen resolutions) at the new location. Certain capabilities might not be available at the new location which will produce their elimination.

4. `createInterface(rolesession1,rolesession2)` allows to recover the role-sessions. As in the previous method, some role-sessions might be no longer relevant.

5. `behaviorChange(roleFigure,state)` allows to recover the state of the actor.

6. After this recovering process the MM is updated via a `locationUpdate(actor1,location2)` method.

5.6.1.4 Evaluation of Mobility in MicroTAPAS

MicroTAPAS is a library that allows actor mobility, but this is not strong mobility, since its control context is dismissed. The actor cannot perform any action during the move.

This library uses a scheme based in two mobility entities: The mobility agent, present in each device, and the mobility manager which is the responsible to perform the move. This entities are organized in a kind of hybrid P2P network architecture, where the mobility manager are the super nodes. MicroTAPAS considers this scheme to share the responsibility of actor mobility (between all the mobility managers).

Its reconfiguration process consider to recover at the new location all the elements an actor had at its old location. It implies to rebind resources (by means of capabilities system), reestablish its relationships (role sessions) with other actors and recover its state.

5.6.2 ChitChat

ChitChat conceives mobility as an action performed by three parties: the sender, the receiver and the moved object. All these parties should play an active role in the move process.

Sender It is the device that sends the object but not necessarily the initiator of the move. It can *decide* how much to move of the object graph.

Receiver This device will receive the moved object. It has to provide the moved object with a number of references so that the moved object can benefit from resource rebinding. It is an object hosted by the receiving host which will have to initiate movement by invoking a move method, hence the receiver determines whether an object may move to itself.

Moved object It is always an active object in ChitChat, since this is the only type of object that can be referenced over the network. An active object has an object graph composed by all objects that collaborate with the active object. An active object *prescribe* how much of its object graph will be moved along, and how much will have to be rebound upon arrival at receiver's location.

5.6.2.1 Move Methods

There is not a unique and structured *move* method in ChitChat. Programmers are free to create this method according to the functionality they expect to have in it. They can choose the name of the method, its arguments (which ones and how many) and its body. The created methods are recognized because they are prefixed by `move` which is a consideration taken in the syntactic system of ChitChat [De Meuter, 2004]. For instance a programmer can choose the name `come` which is included in the code as following:

```
move.come():: { ... }
```

There are some considerations to this flexibility mainly related to the models implemented in ChitChat to deal with objects in open networks (such as the prototype-based and actor models).

Invocation semantics As explained in chapter 4, ChitChat allows to have chains of active objects which can perfectly be distributed over the network. This language is based on prototypes model, and a case of these chains can be delegation chain of an active object. The semantic of move method used in ChitChat indicates that all active objects along this delegation chain, starting from the receiver of the message up until the active object that contains the method will be moved to the machine of the sender of the message for the activation of the move method.

The semantic also dictates that the body of the move method will be always executed upon arrival of the moved object(s) at the destination machine. Hence, this body is considered the action to be taken by the active object immediately after its arrival to its new location.

Marshalling semantics The ChitChat semantics also delimit which part of the internal structure of objects has to be moved (and how) and which part not. The semantics state that an active object has to move with the entity that is representing itself (an active object), its internal definition done by a passive part (potentially composed by Pico tables, passive objects and basic values like methods, numbers and texts), and its queue of communication. In the case of network references pointing to other active objects, these references have to be updated upon arrival at the new machine.

The message that involves a move method on an active object is scheduled in the queue as any ordinary message. This guarantees atomicity in the move action. After the updating of the location, the body of the move method is executed on the new machine. This body may perform resource rebinding based on the parameters it has bounded.

Parameter passing This is a way to make sure that acquaintances are moved correctly: the only way an active object would be able to access a resource, which is in the new location, is to receive a reference of this resource by the sender active object. This is a consequence of message passing. (Note that this operation can properly be done in P2P architectures, whereas it rejects mechanism in which references are obtained from a *central authority server*). The parameters in move methods can be used for this purpose. *By calling a move method with references to local resources, a move method can drag an object to the site of the sender and install references to the local resources at the right spots in the moved object* [De Meuter, 2004].

5.6.2.2 Implementation of the Mobility Model

De Meuter has summarized the ChitChat mobility model as some details about its implementation in the following steps:

1. Move method is invoked by a sender.

2. If sender resides on remote host, the receiver's device prepare the receiver for movement. It implies to stop the execution thread of the active object to be moved, which will enter in a *moving* state. In this state the active object will not execute any message but it will still be able to receive them.

3. Pass the active object by copy to sender machine.

4. Reinstate the active object at the new location, switch its state to *local* and schedule the move method for execution.

5. Notify the original active object and ask it to forward all messages received during movement. It will enter in a *remote* state (becoming a network reference object) and will send those messages to the new active object. Finally the original active object is discarded.

5.6.2.3 Evaluation of Mobility in ChitChat

Active object mobility in ChitChat is a lightly structured process performed by three entities: the (receiver) device that will receive the moved object and contains an object that invokes the move method; the (sender) device that will send the moved object and decide how much to move of the actor; and the moved object itself.

The ChitChat move process occurs transparently for the ChitChat programmer. He only has to create a higher-level move method.

This mobility model ensures that the moved actor never loses its capacity of receiving messages during the move. Only its processing capacity is stopped and restarted when it arrives to the new device. This actor neither loses the order in which the messages were received.

5.6.3 SALSA

SALSA is an actor-based language that provides a mechanism to move *universal actors* around *theaters* (see the full description of these concepts in chapter 4). Like active objects in ChitChat (in previous section), universal actors in SALSA move in response to an asynchronous message requesting migration to a specific Universal Actor Location (UAL) (in order to avoid that the actor processes any other messages during migration). This message is `migrate(ual)`.

Varela describes the SALSA migration mechanism from the perspective of the arrival and departure theaters that will be explained further on. The figure 5.3 shows both theaters before and after the move.

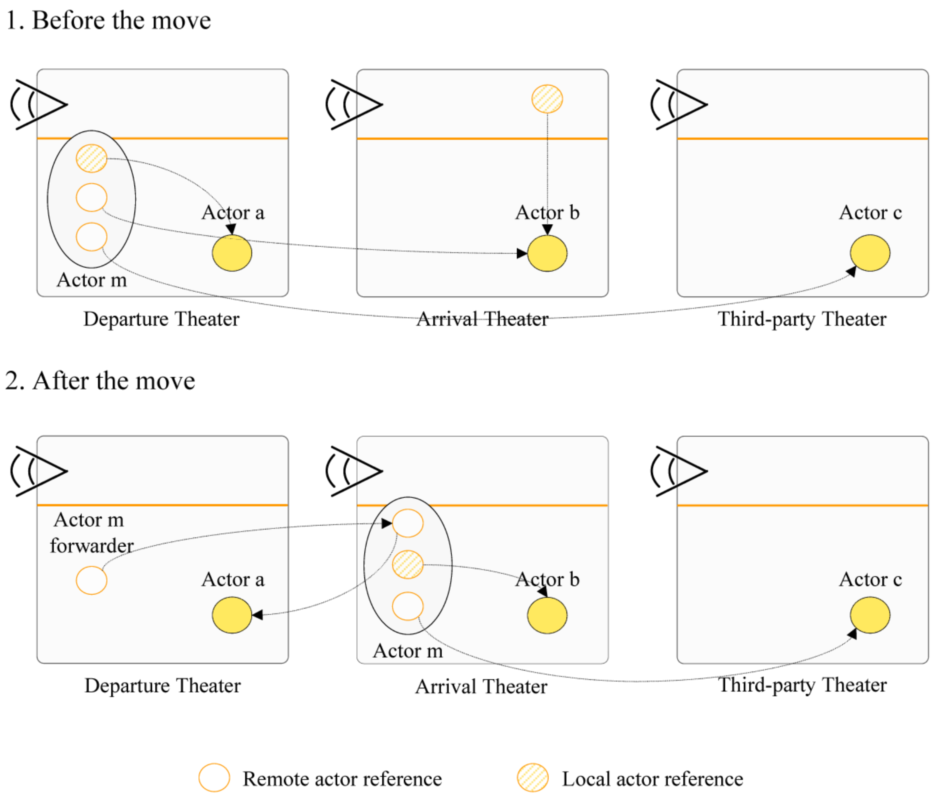


Figure 5.3: Mobility in SALSA

5.6.3.1 Arrival Theater

The RMSP server found in each theater (described in chapter 4) provides a generic input gate for incoming *SALSA-generated Java* objects and a mechanism to receive them.

Part of this mechanism varies according to the type (or class in the context of the java-based SALSA language) of incoming objects, which can be messages and universal actors[1]. Another part is common for all these types. This last part is related to the actor references contained in incoming objects, that are updated by RMSP using the same criteria (see figure 5.3):

- If the UAL of an actor reference points to the current theater, the UAL is updated with the current SALSA reference of the actor found in the RMSP server hashtable.
- If the UAL for the actor reference points to another theater, the reference remains unchanged (as a remote actor reference).
- Every actor reference has a `local` bit indicating whether the reference is pointing to either a local or remote actor. This bit is also updated by the RMSP.

The reception continue in different ways depending of the type of the incoming object, as following:

Reception of a Message the target actor in the message object (after RMSP updates all actor references) has a valid internal reference pointing to the target actor in the current theater. This allows this message to be placed in the mailbox of such actor. If it has moved in the mean time, it leaves a *forwarder* actor at its place. Then, the RMSP server at the new location of the target actor is contacted and the message sending process gets started again.

Forwarder actors are not guaranteed to remain in a theater forever. If RMSP hashtable does not contain the entry for the target actor's UAL, the naming service will look for the new location and if it is found the message is forwarded to the new target actor's location (the process of looking for moved target actors is repeated only 20 times, after what the message is returned to the sender actor as undeliverable).

Reception of a Universal Actor The RMSP hashtable gets updated with an entry mapping the new actor's relative UAL to its recently created internal reference. The actor is restarted locally.

SALSA allows to move a group of actors at once. These group known as *casts* [Varela, 2001] are units for coordination purposes[2].

[1]In early versions of SALSA there were also messengers [Varela, 2001], special actors that migrate with the purpose of carrying a message from a theater to another, but they are not anymore mentioned in newer versions.

[2]*Casts* is a concept used in the hierarchical model implemented in SALSA. The current version of this language has replaced this model by a reflective stack model [Varela, 2001].

5.6.3.2 Departure Theater

An actor that leaves a theater implies the following actions:

- Its state is serialized by the departure theater and moved to the new location.
- The current SALSA internal reference to the actor is updated to reflect the new UAL and its `local` bit is set to false. Thus, the internal reference becomes a *forwarder* actor (explained in the previous section).
- Finally, the (distributed) naming server containing the UAN of the moved actor gets updated with the new actor location.

5.6.3.3 Mobility Implementation

From the point of view of a SALSA programmer, Mobility is implemented as following:

```
TravelAgent a = (TravelAgent)
  TravelAgent.getReferenceByName("uan://myhost/ta");

a <- migrate("rmsp://yourhost/travel") @
  a <- printItinerary();
```

This sample code can be summarized in three steps:

1. Obtaining an universal actor reference.
2. Sending the `migrate` message to the universal actor.
3. Optionally including a continuation that will be executed upon actor's arrival at new location.

At the implementation level, the internal migration steps were summarized by Varela as follows:

1. Updating the naming service to reflect the actor's new locator.
2. Serializing the actor's state to the new theater.
3. Updating the actor's references to local resources (called *enviroment* actors in SALSA).
4. Updating the theaters' meta-data (when the stack meta-model is used).
5. Restarting the actor's thread in the new location.

5.6.3.4 Evaluation of Mobility in SALSA

SALSA move process is initiated by a asynchronous message `migrate(ual)` (similarly to ChitChat in the previous section). It allows the concatenation of a continuation which will be executed at the arrival location. This continuation can be used as a higher-level coordination instance after the move.

This mechanism implies direct interaction with the universal naming service protocol (UNAP). It means, SALSA developers should know UALs and UANs of the actors implied in this move process.

The SALSA mobility mechanism concentrates mainly in the network reconfiguration process. It implies the reference updating strategy performed by the RMSPs found at the departure and arrival devices, and the updating of references to local resources (environment actors).

5.7 Conclusion

This chapter has described the strong mobility of code in the scenario of the mobile open P2P networks. First, it identified the reasons for mobility in mobile open networks. The first of them was related to the traditional reason for code mobility: move computations towards desired resources or away from scarcity. This reason applies properly for the context of mobile open networks due to the connection volatility and resource scarcity of their participants (mobile and embedded devices). The second reason is based in AmI vision and the way in which technology will surround the people. It will allow to create applications that hop from one device to another in order to ensure the availability of their services for the users.

Respect to the requirements, these were related to the properties of devices (connection volatility, scarce resources and heterogeneity). All of these properties imply a different consideration for the code mobility.

This chapter defined different types of code mobility according to their computational context. It was explained that strong mobility of code has advantages over the other types, mainly because it implies that moved code (application) can continue working during their move, and it can rebind resources at new locations.

Respect to the languages implementing code mobility presented in this chapter, the following properties summarize their properties:

- Identification of destiny and source locations. Devices are usually recognized by an entity that resides inside of it.
- Identification of the moved object. It has a unique identification (and location) over the network.
- Identification of a method that initiates the move process. In all the actor - based approaches it corresponds to a message received either directly by the actor or by a entity that will perform the mobility.

- Performance of network reconfiguration after the move. All approaches considered a network reconfiguration approach after the arrival of the actor to the new location. In the best cases this is a decentralized mechanism (using a P2P scheme).
- Performance of intra device reconfiguration. Some approaches provide a intra device reconfiguration mechanism that enables the moved actor to rebind resources at the new location.

Chapter 6

Strong Mobility of Ambient Actors

6.1 Introduction: The AmbientTalk Mobility Model

The model for mobility presented in this work is aimed to provide support to ambient actors in dynamically reconfigurable environments, such as open networks. These networks can change due to either devices or computation entities (such as actors) moving around it. These movements can be produced involuntarily (for instance, because of the connection volatility of devices, which also affects the actors inside of these devices) or voluntarily (predefined heuristics that force actors to move). Thus, the mobility model in AmbientTalk is conceived to support network reconfigurability in terms of both device and actor mobility produced either intentionally or unintentionally. But the goal of supporting network reconfigurability is still quite vague, since it can be matter of network architecture decisions, like those taken in the different cases of P2P applications described in chapter 3. The P2P architectures are part of the context of this work but not of the proposal, since they deal with lower levels of decisions like peer discovery and routing protocols[1]. Nevertheless, this vision assumes that peers correspond to devices on a network.

The AmbientTalk distribution model also uses a P2P-based scheme; however, in this case, peers are assimilated to ambient actors capable to discover and communicate among themselves. As explained in chapter 4, peers use an actor-lookup based on the patterns they offer. These patterns can be understood as the services provided by the actors, themselves related to their behaviors. It means that an open network based on ambient-actors is nothing more but a set of services provide by them. Thus, the support to the inherent dynamism of an open network is specially related to ensure, as much as possible, the availability of these services over the network.

[1]It is not the goal of this work to propose a new low-level communication layer with a P2P algorithm different to the one actually used in AmbientTalk (a broadcast-based P2P algorithm), like a DHT-based P2P algorithm, which would be interesting if looking for efficiency.

The mobility model presented in this chapter deals with device and ambient actor mobility, which both increase the availability of the services offered over a network.

6.2 Device Mobility

As explained in the chapter 2, devices have volatile network connections because some of them may be mobile, thus directly influencing their connection quality as they move about. Consequently, the actors inside of these devices, communicating with other actors from other devices, have volatile relationships with each other as well. When a connection is lost, devices can either restore the connection later, or not be reconnected anymore to that network. This mobility model provides a mechanism for making actor communications less vulnerable to these scenarios. This mechanism is based on *Ambient References* which will be explained in the following section.

The ambient-oriented model implemented in the AmbientTalk kernel, providing reified actor communications and acquaintance management mechanisms, is already conceived to deal with device mobility (see chapter 4). However, the abstractions found there to create programs are closely related to the mechanisms themselves. It means that programmers are forced to deal with mailboxes and reflective methods created for these mechanisms in their programs. The idea of the mobility model presented in this work, is to offer abstractions (already dealing with these ambient-oriented mechanisms) that allow programmers to build software using more high-level programming constructs. The programmer would not look anymore at *how* actors should communicate with one another in the system[2].

6.2.1 Ambient References

An Ambient Reference is an abstraction built in AmbientTalk that represents, at one device, an ambient actor located in another device. Its purpose is to provide reliable communication between actors residing at the device where the ambient reference is created, and the remote actor to which this reference is representing. As explained in chapter 4, every communication in AmbientTalk involves a *pattern* provided by an actor and required by another one. An ambient reference can be also expressed in these terms. It is an intermediate entity that requires a pattern (service) provided by an actor in a remote device. An ambient reference is a pointer which, instead of pointing to an actor address, points to a pattern, which decouples the pointer from one specific actor. The figure 6.1 depicts a ambient reference providing a pattern (represented by a geometric figure in this case).

The goal of using Ambient References is to abstract away the handling of service discovery and the results of network failures.

[2]It does not preclude access to the components and methods offered by the AmbientTalk kernel, if required.

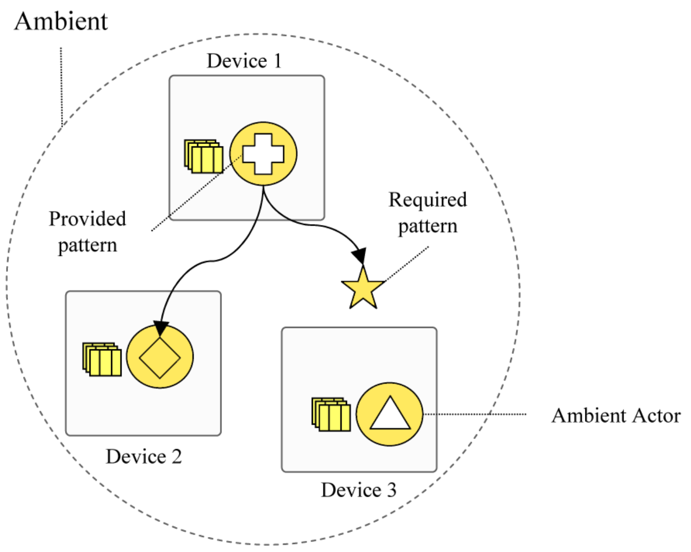


Figure 6.1: Ambient Reference

6.2.2 Implementation of an Ambient Reference

An ambient reference is implemented in AmbientTalk programming language as an actor that contains a required pattern (`itsPattern` field), a remote actor providing the pattern (`itsProvider` field), and a set of methods defining its behavior. The remote actor object is explained in section 6.3, it is similar to a SALSA actor reference [Varela, 2001], which is a lower-level reference than ambient references.

The following piece of code shows all the implementation of a basic ambient reference.

```
ambientRefBehaviour::view({
  itsPattern  : void;
  itsProvider: void;

  cloning.new(aPattern)::{itsPattern:=aPattern};

  init()::required.add(itsPattern);

  joined(aResolution)::{
    display(provider(aResolution), " joined on ",
            pattern(aResolution), eoln);
    required.delete(pattern(aResolution));
```

```
      if(is_void(itsProvider), {
           itsProvider:=provider(aResolution);
           inbox.asVector().iterate({
             outbox.add(setMsgTarget(copyMsg(el),
                                          itsProvider)) })
         })
    };

    getProvider()::itsProvider;

    in(aMsg)::
      if(not(or(is_void(itsProvider),
                    containsBehavior(msg.getName()))),
         outbox.add(setMsgTarget(copyMsg(aMsg),
                                            itsProvider)))
  }).futuresMixin();

  ambientRef(pattern)::
              actor(ambientRefBehaviour.new(pattern));
```

The function `ambientRef(pattern)` in the last line of the code is all a AmbientTalk programmer needs to know to create an ambient reference. It will be used as following:

```
  myRemotePrinterActor: ambientRef("print service")
```

This line creates an ambient reference that represents an actor over the network providing the pattern identified by the *print service* string[3]. Note that since an ambient reference is an ambient actor itself, the rest of the actors in the device can start immediately to send messages to this ambient reference upon its creation. An Ambient Reference is a local proxy for a remote actor, which may or may not yet be discovered. But because it is a local actor, it can receive messages regardless of the discovery state of the ambient reference

The five methods of the ambient reference behavior have the following meaning:

- `cloning.new(aPattern)` is the Pico-based way to create an object (an actor behavior in this case) [De Meuter et al., 2004]. This constructor receives the required pattern as parameter.

- `init()` initializes the actor by putting the pattern required in this ambient reference in its `required` mailbox.

[3]Future versions of AmbientTalk will allow this identifier to be any object, not only a String.

- `joined(aResolution)` is a reflective method provided by AmbientTalk that is called whenever a resolution is received in the `provided` mailbox of the ambient reference. Remember that a resolution is a tuple containing a pattern and a remote actor providing this pattern. The implementation of this method deletes the pattern from the `required` mailbox, sets the remote actor (itsProvider) and forwards to the remote actor all messages received by the ambient reference (in its `in` mailbox) before the resolution was gotten.
- `getProvider()` returns the remote actor.
- `in(aMsg)` is another reflective method that is invoked whenever a message is received in the `in` mailbox of the ambient reference. This method is implemented to ensure that every message received by the ambient reference will be forwarded to the remote actor if it is already discovered and joined. Otherwise, messages will remain in the `in` mailbox and will be forwarded upon the joining reflected in the `joined` method.

The actor of the device that contains the ambient reference can send messages to the remote actor represented by this ambient reference even if none of the two devices implied in this communication are connected to the network. It is because the ambient reference is an actor itself provided with reified communication (explained in chapter 4); reified communication ensures that messages sent from an actor to another no longer connected will remain in its `out` mailbox until the target actor joins the network again.

Thus, this ambient reference complies with the first scenario identified above, related to devices that leave the network for a while and join it again afterwards. But the second scenario is not accomplished by this type of ambient reference. The ambient reference pointing to a remote actor residing at a device that does not join the network anymore (or that is destroyed or moved to another device), will not be useful anymore and the messages received for it will be lost. To solve these problems, other types of ambient references have been created. These will be described in the next section.

6.2.3 Types of Ambient References

Three types of ambient references have been created: strong, semi-strong and weak references.

Strong ambient reference This is the original type of ambient reference explained in the previous section. It is called ”strong” because, once it binds to a specific actor, it can never ”rebind” to another actor. It can be used for the case in which it is required to maintain the communication with a specific actor that provides certain pattern. Once this actor is matched with the ambient reference, none other actor providing the same pattern will be accepted in its place, even if the actor is not connected to the network. A strong ambient reference will get problems when the remote actor to which it points is destroyed or even just moved. This will be explained in the following section.

Weak ambient reference This type of ambient reference does not care about the actor providing the required pattern, but only about the pattern itself. It may rebind to any actor providing the pattern required by the ambient reference. This means that this reference will join a remote actor and after its departure, the ambient reference will look again for the pattern in other actors.

Semi-strong ambient reference This type of ambient reference is useful to follow an actor around the network. Once it ”binds” to a certain actor providing a certain service, it only allows rebinding to the specific service (pattern) offered by this actor. This ambient reference not only matches the pattern but also stores the name of the first actor found, providing the required pattern.

The distinction of these types of ambient references is relevant when working with mixing cases of mobility (actor and device mobility). These scenarios will be explained in section 6.4. The role of ambient references in dynamically reconfigurable networks.

6.3 Strong Mobility of Ambient Actors

The second concern in the AmbientTalk mobility model is related to the strong mobility of ambient actors. It implies the following:

- This model provides strong mobility of ambient actors by making them available even during the time in which they are moving to a new location.
- This strong mobility mechanism performs a transparent P2P-based reconfiguration of the communication relationships with other actors upon the arrival of the moved actor to the new location. It is to be said this reconfiguration is transparent because actors do not lose their relationships at any time.

It has been explained in this work that strong mobility is related to move a computation with its data and control contexts (see chapter 5). In terms of actors, the data context can correspond to its *state* (which is considered in the most of the traditional mobility approaches), while the control context can be interpreted as its *running process*. In the actor model defined by Agha [Agha, 1986] the running process was in charge of receiving and executing incoming communications. The ambient actor model extends its responsibilities to providing reified actor communications and to managing the acquaintance of its network environment (based on the pattern-lookup service).

Regarding these responsibilities, the strong mobility mechanism provided in this model takes the following considerations into account when an ambient actor demanded to move:

1. An ambient actor will continue receiving messages from other actors during all the move process. However, at some moment of this process (this moment is identified in the description of steps of the move process in section 6.3.2) the received messages will not be processed until the actor restarts its execution (its thread) at the new location. In spite of this, messages will always be processed at the same order they were received. A similar solution is present in the ChitChat mobility model (chapter 5).

2. An ambient actor will continue providing the reified communication traces. Note that at the moment in which the actor is not processing any message, the responsibility of this mechanism will be limited just to receive external messages in the `in` mailbox.

3. Finally, an ambient actor will comply with the acquaintance management responsibilities. During the moment of the move process mentioned in the first point, in which the actor does not process messages, it will continue providing its patterns to other actors over network, but it will not require any pattern (service) from abroad. This actor will look again for required patterns only when it restarts its execution at the new location.

Local and Remote Actors It is important to say that ambient actors are already transmitted over the network, before the implementation of this strong mobility mechanism. It is referred to pass actors by reference, i.e. that when actors are about to be passed over the network, a remote reference is passed instead. Hence, there is no form of actor mobility whatsoever, an actor never moves from one machine to another. This previous transmission mechanism was included into the AmbientTalk kernel for supporting the pattern-lookup service and the message-passing communication system. In both cases actors are passed as parameters of remote communications (actor messages or discovery results). In none of these cases it is necessary to transmit the actor itself with all its components (state, behavior, thread and mailboxes). A lighter version of it, containing its location and its identification, is good enough (as a SALSA actor reference with its UAL and UAN [Varela, 2001]). This light version of the actor is called *remote actor*. The original version of the actor working at some device is known as *local actor*. The transmission mechanism (implemented using Java serialization[4]) ensures the correct replacement of the different versions of actors in the following way:

- A local actor is replaced by its corresponding remote actor when it is transmitted[5].

- A remote actor arriving to its original location is replaced with the corresponding local actor.

[4]Remember that the current version of AmbientTalk was built in Java, in the J2ME platform.

[5]AmbientTalk uses hashtables to ensures proper actor version replacements.

It is not the goal of strong mobility of ambient actors to pass local actors as remote ones to other devices; but it is to move them in such a way that they arrive to new locations as local actors too. Nevertheless, the strong mobility mechanism takes advantage of this already existent transmission mechanism to perform the move process (see section 6.3.2).

6.3.1 Strong Mobility of Ambient Actors in AmbientTalk

The ambient actor mobility is a lightly structured process from the point of view of the AmbientTalk programmer, in the same spirit of Emerald and ChitChat (see chapter 5). It considers a primitive which can be put inside of a more sophisticated move method, if required. The primitive is called `moveActor` and takes two arguments: a remote actor (explained above in this section) and an actor's message that will be processed right upon the restarting of the actor at the new location, which can be used for coordination purposes like in SALSA (see chapter 5).

Note that a move method in an actor behavior enables this actor to be requested to move by any device over the network that holds an ambient reference pointing to this actor (by sending it the move message).

The following code shows the implementation of a possible move method using the `moveActor` primitive.

```
travelerActorBehavior:object({
    ...
    moveWhere(aRemActor):: {
        moveActor(aRemActor,
                  createMessage(thisActor(),
                                thisActor(),
                                "foo",
                                [x]))};
    ...
    foo(x):: { ... };
    ...
});
```

The method in this case is called `moveWhere` and receives an remote actor as a first argument. It can be understood as *move this actor to the location of the actor represented by `aRemActor`*. The second argument is a message[6] to itself invoking its `foo(x)` method. Note that since this method is part of the behavior of this actor, every device that contains an ambient reference to this actor can send the *moveActor* message. It means, this actor can be demanded to move from any place.

[6]`createMessage()` is a function that receives as parameters the sender, the receiver, the name of the message and the arguments. Note that in this case the message is sent to itself but there are no restrictions about the receiver of the message. It can be anybody.

6.3.2 Implementation of Strong Mobility of Ambient Actors

The move process implemented to strongly move ambient actors in AmbientTalk has four main steps. This process is complementarily fulfilled by the departure, the arrival device and the moved ambient actor itself (like in ChitChat [De Meuter, 2004]). These steps are delimited by four communications sent in order to accomplish with the move. All these communications (except the first one that corresponds to the invocation of the primitive used at AmbientTalk programming level explained in the previous section) correspond to messages sent between devices, known in AmbientTalk as *commands* (these are used only inside of the AmbientTalk kernel). The four communications are the following:

The `moveActor` primitive As explained above, this primitive should be included in some move method built at AmbientTalk programming level. This primitive receives as parameters a *remote actor* which device will be the new location of the moved actor, and a *continuation message* that will be the first message to be processed once the ambient actor is restarted at the new locations (similar to a remote continuation after an actor migration in SALSA [Varela, 2001]).

The `moveActor` command This is the first command sent from the departure device to the arrival one. It receives as parameters the *local actor* and the *continuation message* sent by the AmbientTalk programmer in the previous primitive. Remember that passing a local actor as parameter implies that it will arrive as a remote actor to the new location (by the actor transmission mechanism). This parameter is only carried for coordination purposes in the command chain of the move process, as explained later in section 6.3.2.4.

The `resultMoveActor` command This command is sent back from the arrival device to the departure one. It receives as parameters the *old local actor* and the *new local actor*. The *old local actor* parameter is currently a remote actor as explained above, but it will become a local actor at the departure device. The opposed case is for the *new local actor* parameter (this is the newly instantiated actor at the arrival device) which is currently a local actor and will be converted in a remote actor at the destiny location of this command.

The `moveActorContents` command This is the last command communicated in this process. It receives as parameters one more time the *old local actor* and the *new local actor* (in the correspondent states according to the actor transmission mechanism), and the *behavior* and *mailbox contents* of the old actor.

As mentioned above, strong mobility of ambient actors has a process composed of four steps created to prepare the ambient actor to move, reinstantiate it at the new location, move properly the components of the actor, and finally restart it at the new location.

6.3.2.1 Preparing the Ambient Actor to move

This preparation implies two parts: checking the move preconditions and starting the move process by sending the first command from the departure to the arrival device.

Checking preconditions Upon the invocation of the `moveActor` primitive, the ambient actor, supported by the departure and arrival devices, checks a set of preconditions intended to validate the current demand to move[7] (like in ChitChat or in SALSA). These preconditions can vary from simple validations (for instance, to validate that departure and arrival devices are not the same) to more sophisticated ones intended to ensure the adequate environment for the incoming actor. In this last sense, a meaningful way to represent the devices is relevant. Such is the case of the *capabilities system* provided in MicroTAPAS [Luhr, 2004], in which the capabilities are the services provided by the devices (like hardware resources). Another similar example is the inclusion of *environmental actors* in SALSA. In both approaches these abstractions not only support the work of actors in devices, but also are considered decision factors to move (or not) an actor to the demanded location. This instant of checking preconditions let the final decision to move the actor to itself, which is likely the sprit of the sentence *move method as hints* described in Emerald [Briot and de Ratuld, 1988].

Sending the `moveActor` command If the preconditions are properly fulfilled, the departure device sends the `moveActor` command. It will create a deactivated ambient actor at the new location, which means that the new actor will not process messages yet (nevertheless, it can receive them already). At this moment, some actor mobility approaches like the one found in ChitChat, stop the execution thread of the old local actor at the departure location, letting it only with the capacity to receive messages but not to process them. This decision seems not to be the most adequate one if the goal is to maintain as much as possible the availability of the moved actor and its services (patterns) offered to the network. The reason discussed in this work for stating this, is related to the volatility of devices in open networks. If the arrival device gets loss of connection right after it receives the `moveActor` command, then the ambient actor at the old location will be lost too (in terms of its process capacity). It is because the arrival device will not be able to send the second command which enables the continuity of the move process back to the departure location.

Letting the moved actor to work at the old location during this step of the move process (even with its processing capacity), implies that none of its components (state, behavior and mailboxes) should be moved to the new location. That is because this action implies copying the actor components to the arrival device,

[7]In this implementation these preconditions are expressed at the level of AmbientTalk: they are simply boolean tests represented by AmbientTalk expressions.

which could produce an inconsistency between the status of these components at the time in which this first command is sent, and the same status at the time in which the actor will restart to work at the new location (The following steps will argue why these are two different times). One example of this inconsistency can be seen in the behavior of the actor. According to the *history-sensitive behavior* mechanism described in the traditional actor model (see section), the behavior of an actor is always expressed as a function of its incoming messages. In other terms, the execution of a message can change a current behavior. Since the actor at the old location will continue to process messages, its behavior could have changed during the time occurred between the `moveActor` command is sent to the arrival device and the actor restart its work at that new location. It will not make sense to process a message with a wrong behavior.

The same analysis can be done in the context of the state or mailboxes of the actor, which could change during that time too.

Taking into consideration the previous discussion it is possible to conclude that the `moveActor` command will instantiate an empty version of the moved ambient actor at the new location. The following steps will explain how this action will ensure the strong mobility of this actor.

6.3.2.2 Reinstantiating the Ambient Actor

This step starts when the `moveActor` command is received by the arrival device and finalizes when it sends back the `resultMoveActor` message. The latter is sent in the context of the execution of the former.

Creating an ambient actor The execution of the `moveActor` command considers first the creation of an ambient actor which will continue the work of the actor demanded to move. As explained in the previous step, this new ambient actor is currently *deactivated* and *empty*. It is *deactivated* in the sense that it cannot process any message since its execution thread is not running. Nevertheless, it can receive messages already. It is *empty* in the sense that it does not have any behavior and its mailboxes do not contain any message. They will be filled once the content of the old actor at the departure device is sent in the `moveActorContents` command. However, note that mailboxes are ready to receive messages. It means that every actor over the network that gets a reference to this new actor can send messages to it already. The missing point is that this actor is still not published to the network (it is not reachable by the pattern-lookup service). This will be done only once it gets filled (with its state, behavior and contents of its mailboxes) and activated.

Sending the `resultMoveActor` command The last action in this step is to send back the `resultMoveActor` command from the arrival to the departure device. As said before, this command receives as one of its parameters this new local actor which will arrive to the departure device as a remote actor. This reference will be useful to forward the messages received by the moved actor of the old location to the new actor. The next step explains when it is required to forward the messages.

6.3.2.3 Preparing the actor contents to move

This step starts with the reception of the `resultMoveActor` command at the departure device and finalizes when it sends the last command of the chain called `moveActorContents`. This step happens in the context of the execution of the `resultMoveActor` command.

Converting the old local actor into a remote one The reference to the new actor created at the new location, carried by the `resultMoveActor` command, allows the old actor to delegate the responsibility of receiving messages from other actors over the network, to this new actor. It is achieved as following:

- The reference (a remote actor, as explained before) is included to the state of the old local actor.

- A local actor with a reference to a remote actor will imply that all messages sent to it, will be finally received by the remote actor to which the reference is pointing. It does not necessarily mean that the messages need to be forwarded by the local actor to the remote actor, since it implies to have a rerouting mechanism, which is harmful for the reliability of the communications, as explained later in the section 6.3.3.1.

The next step will show that the same mechanism of including remote actor references into local actors can be done in the other way around (local actor references into remote actor) with a reversed result.

Sending the `moveActorContents` command The new actor at the arrival device is now properly receiving the messages sent to the *moved actor* (note that at this moment the *moved actor* is a conceptual term accomplished by two actors). This change happened transparently from the point of view of the other actors over the network. In other words, the actor is being *strongly moved* in terms of its communication relationships. It enables the old actor to move all its components to the new actor, which means, to send its behavior[8] and all the the content of its mailboxes with exception of the resolutions stored in the `joined` and `disjoined` mailboxes, since these resolution will be automatically recovered by the pattern-based lookup mechanism.

[8]A behavior in AmbientTalk is implemented as an object whose fields represent the state of the ambient actor.

6.3.2.4 Restarting the Ambient Actor

This final step starts with the reception of the `moveActorContents` command, and finalizes with the restarting of the processing capacity of the moved actor. This step occurs in the context of the execution of this received command.

Restarting the processing capacity of the moved actor The reception of the components of the old actor found in the `moveActorContents` command, enables the new actor to complete its state in the following way:

1. The behavior of the old actor received with the `moveActorContents` command is copied to the behavior of the new actor.

2. The messages found in the `in` mailbox received with the command, are added to the `in` mailbox of the new actor, in such a way, these messages will be executed before the messages found in the mailbox of this actor (if any), once it starts its execution thread. Remember that the new actor could receive already the messages sent to the *moved actor*, which chronologically were sent *after* the messages found in the `in` mailbox of the old actor (received with the command).

3. The continuation message is added to the `in` mailbox of the new actor in such a way that this message will be the first to be executed once this actor starts its execution thread.

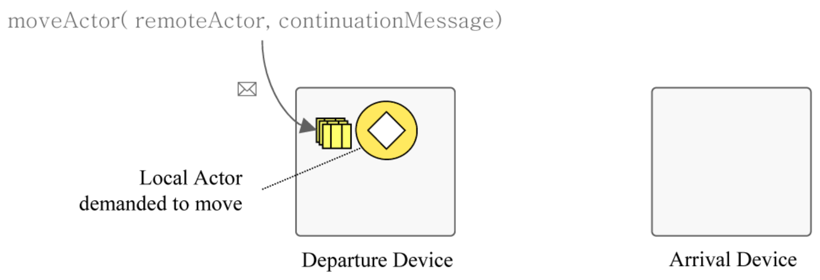
4. The content of the mailboxes `rcv`, `out`, `sent`, `provided` and `required` received with the command are moved to the correspondent mailboxes of the new actor.

After completing the state of the new actor it will start its execution thread, which implies it will be started to process the incoming messages and it will become *reachable* by the pattern-based lookup mechanism.

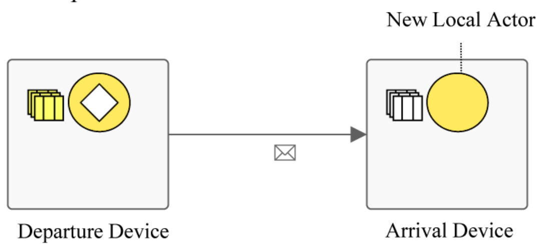
Converting the old remote actor into a local one The move process finalizes updating the remote actor found at the arrival device (if any) that is pointing to the moved actor at its old location. This is achieved by using the same mechanism to convert the old local actor into a remote one, described in the previous step of this process, but in the other way around. More details about these mechanisms are given in the next section, which is related to the network reconfiguration after the strong mobility of ambient actors.

The figure 6.2 depicts all the steps of the move process implemented for supporting strong mobility of ambient actors in AmbientTalk.

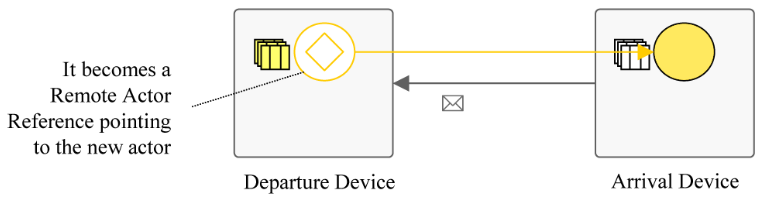
1. Local Actor receives a moveActor message.


moveActor(remoteActor, continuationMessage)
Local Actor demanded to move
Departure Device
Arrival Device

2. Creation of a new local Actor (empty and deactivated) at arrival device. The *moved actor* still works at departure device



3. Change of state of old actor (local to remote). New actor starts to receive messages for the *moved actor*.


It becomes a Remote Actor Reference pointing to the new actor
Departure Device
Arrival Device

4. Move contents (mailboxes, state, behavior) to the new location of the actor. The continuationMessage is put at first place in the inbox mailbox. New actor starts to process messages.

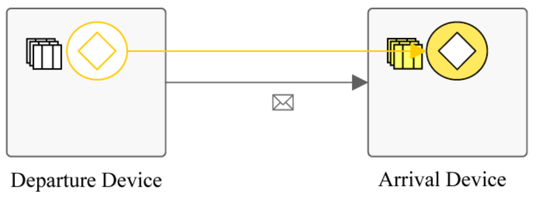


Figure 6.2: Strong mobility of ambient actors

6.3.3 Network Reconfiguration

The mobility mechanism explained above requires network reconfiguration for ensuring the proper work of the system once the move is done. It implies to update all the relationships the moved actor have with other actors over the network. Remember that this mobility process is aimed to be a P2P-styled process. As such, this process should comply with the properties of these architectures. It will be evaluated in section . The network reconfiguration step can be considered part of the self-organization capacity of the mobility model. As such, it occurs transparently for the programmers (they do not deal with network reconfiguration at programming level). A similar solution can be found in SALSA [Varela, 2001].

Three different cases requiring network reconfiguration after moving an actor have been recognized. These are the reconfiguration produced at the departure and the arrival device and also at any third-party devices which can contain actors communicating with the moved actor. As it is explained in the following three sections, the network reconfiguration already starts while the actor is moving. Figure 6.3 depicts the three devices implied in the network reconfiguration.

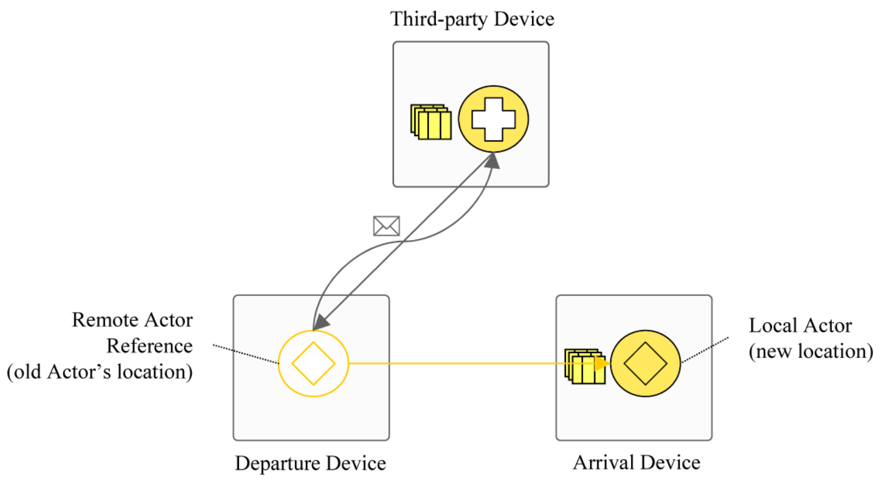

Third-party Device
Remote Actor Reference (old Actor's location)
Local Actor (new location)
Departure Device
Arrival Device

Figure 6.3: Network reconfiguration

6.3.3.1 Reconfiguration at the Departure Device

This network reconfiguration is related to all the communication relationships that the moved actor has with other actors residing at the departure device. The reconfiguration in this case implies the following steps:

1. The reconfiguration at the departure device starts in the third step of the move process (section 6.3.2.3). After this device receives the `resultMoveActor` command, it converts the old local actor into a remote one by putting the reference pointing to the remote actor inside of it, carried by the command.

2. Once the old actor is converted to a remote one, all messages sent to it will be *forwarded* to the remote actor at the new location. It is valid for both local and remote relationships the moved actor have with other actors. It can be understood as a *message-rerouting* mechanism which is a usual P2P solution (as in SALSA). However, this mechanism can deteriorate the reliability of the actor communications. An actor that sends a message to another one, will suppose that the message was properly received by the actor, as soon as this message arrives to the device containing the receiver actor. It will not change when using a message-rerouting mechanism. In such a case, if the receiver actor is moved to another device, the message will be rerouted to the new location, without any notification to the sender actor about this rerouting. It could seem very useful for fixed networks. But it is not the case for open networks where actors are exposed to the connection volatility between the different devices. If the device in charge of rerouting the message is no longer available, the message will never arrive to its final destination, and the sender actor will never get aware of this.

   The following two sections will propose an alternative (P2P) solution for this problem. In the meantime, it is to be said that message-rerouting mechanism (achieved by inserting the remote reference into the old local actor) will be only useful for the actors communicating to the moved actor from its old location, it means, at the departure device. That is why this system will be called *local rerouting*.

6.3.3.2 Reconfiguration at the Arrival Device

This network reconfiguration is related to all the communication relationships that the moved actor has with other actors residing at the arrival device. The reconfiguration in this case implies the following steps:

1. The reconfiguration at the arrival device starts in the last step of the move process, after the reception of the `moveActorContents` command. In this step, the remote actor that represented the moved actor when it was at the old location, is updated.

   It was explained in section 6.3 that the only additional ways (to the one presented in this move process) by which an actor arrives to a new location, is when it is carried by a message or when it has a pattern required by another actor residing at this new location. Both cases are handled by the AmbientTalk transmission mechanism that ensures to pass a light version of such actor (called remote actor) through the network. Thus, these cases are the only ways by which a remote actor appears at that new location.

Coming back to the case of the moved actor, if there is already a remote actor at the arrival device representing this moved actor, the remote actor will be updated.

2. It is updated by putting inside of it a reference to the new local actor (representing the moved actor). It is done to ensure that all communications between moved actor and the rest of the actors at the arrival device, become local. It means, it will not be sent to the old location and *rerouted* back to the new one.

For ease of reference, this part of the network reconfiguration mechanism will be called *local communicating* reconfiguration.

6.3.3.3 Reconfiguration at a Third-party Device

This network reconfiguration is related to all the communication relationships that the moved actor has with other actors residing at any *third-party* device which is neither the departure device of the moved actor nor its arrival one. The reconfiguration in this case implies the following steps:

1. An actor residing at a third-party device, as explained above, sends a message to the moved actor at the old location, since it does not know anything about the change of location of the moved actor. In such a case, the sender will be notified by the arrival device that the receiver actor moved. This notification contains the information about the new location of the actor.

2. Once the sender actor receives this notification, it *re-sends* the message to the actor in its new location.

Rerouting and updating Note that this mechanism named *updating by notification* avoids the rerouting of messages sent by actors that are not in the departure device. It is a good achievement for the reliability of the system. However, the discussion about reconfiguration is not included with this solution. It could be possible to have an intermediate solution between the message-rerouting mechanism and the one proposed in this work. This third solution lets the departure device to assume the responsibility of finding the current location of the moved actor. This seems very similar to the solution described above. However, this third mechanism makes the difference in the case an actor moves more than once over the network. In such a case, this device will look for the moved actor over the network, instead of immediately sending back a response to the sender actor of the received message. In this mechanism, it could be possible that the device containing the moved actor sends back the notification, which would reduce the number of communications implied in this lookup mechanism, turning it more efficient and reliable than the first two solutions.

Vulnerability of the *updating by notification* mechanism The network reconfiguration mechanism at a third-party device proposed in this work is less vulnerable to the connection volatility of devices than the message-rerouting mechanism, but it is still vulnerable. The loss of connection with the departure device implies that actors over the network that did not update their references to the moved actor (by sending a message to the moved actor when the departure device was still on the network), will not be able to access this actor anymore. For solving this problem, it is necessary to take into consideration all the mechanisms introduced to deal with device and actor mobility. The following section will show that ambient references already taking advantage of the reified communication and pattern-based lookup mechanism of AmbientTalk, work properly even with actors moving over the network.

6.4 Mixing Device and Ambient Actor Mobility

As mentioned in the previous section, mobility of actors in open networks is not a phenomenon that occurs isolated. This process can be affected by the mobility of the devices implied in the actor move. Nevertheless, not only the two types of mobility can be mixed but it is also possible to handle the different types of mobility in a uniform way. The following situations are included in this chapter as a way to close the discussion about the actor move process itself in this work. Further cases of mobility described in the next chapter are intended to show the application of actor mobility in different scenarios in open networks.

6.4.1 Loss of Connection During the Move Process

As largely explained in the previous section the move process has four steps. Devices involved in this process can become disconnected during the execution of these steps. The following analysis will evaluate, for each step, the condition of the communication relationships between the moved actor and others found in three different locations: the departure, the arrival and the third-party device. More specifically, the evaluation of the communications is done for the particular case of *actors from the three mentioned devices sending messages to the moved actor*.

In each step, the loss of connection of the departure and the arrival device will be separately evaluated (note that the disconnection to the network of a third-party device is not relevant for the fulfillment of the move process).

As said before, analyzing mixed mobility scenarios also implies the possibility of mixing mechanisms created to deal with the different types of mobility. At the very beginning of this chapter three types of ambient references were introduced. The ambient references concept already takes advantage of the reified communication and pattern-based lookup mechanisms, included in AmbientTalk to deal with device mobility. The following analysis will show the role of ambient references in the reconfigurability of a network during the actor move process. Note that after all the definitions of models, mechanisms, process and steps described in this work, the only two *extra* forms that will be needed to work with *mobile actors supporting reconfigurable applications in open P2P networks* are the instantiation of an ambient reference and the `moveActor` primitive included somewhere in its behavior.

6.4.1.1 While preparing the ambient actor to move

As explained in the previous section this step only consists of performing a mobility precondition check and sending the first command from the departure to the arrival device of the actor to be moved. As such the only effects will be related to the mobility of devices. Thus, this scenario is matter of the properly work of the ambient references.

Loss of connection at the departure device The loss of connection at this device implies that the moved actor is not available anymore. Thus, it is a case only to evaluate the work of the ambient references. All of the ambient references are provided with reified communications. Then, actors residing at the devices of a ambient reference pointing to the moved actor, will continue to send messages to it. After the disconnection of the moved actor, notified by the acquaintance management mechanism, the three ambient references will behave differently.

- The strong ambient reference will wait until the moved actor get reconnected to the network at the same device in which it was discovered (the departure device in this case). If this happens, this reference will send it all the enqueued messages. This kind of *fidelity* is due to the simple fact that it does not do anything after the disconnection of the remote actor already registered in its state. It is not the case of the other types of ambient references.
- The semi-strong ambient reference will wait until the moved actor get reconnected. After the disconnection of the moved actor, this reference clear the `itsProvider` field its state. However it has stored its name in other field (`itsProviderName`). It implies that this type of reference will wait for the moved actor, but it will be also able to find the moved actor in another device, if it is the case. This is a kind of *smart fidelity*.
- Finally, the weak ambient reference will not wait for the moved actor. Upon disconnection the moved actor, it will immediately start to look for another actor providing the required pattern registered in its state (it clears the `itsProvider` field and does not store any information about the actor).

Loss of connection at the arrival device As this device does not have any responsibility until the moment in the move process, there is not too much to say about this case. The only change of state perceived in this case will be in the ambient references residing at this device. They will behave in the same way described in the first case.

6.4.1.2 While reinstantiating the ambient actor

This step corresponds to the reception and execution of the first command sent by the departure to the arrival device. This execution at this second device considers the creation of an empty ambient actor and the sending of the second command to the departure device. Although not that critical, the consequences of stoping abnormally can start to be identified in this step.

Loss of connection at the departure device The disconnection of the departure device at this step will not imply any different consideration for the third-party device to the ones described in the previous step (ambient references accumulating messages and waiting or looking for other actors according to their behavior). For the arrival device instead, the disconnection of the other device will imply to get an empty and unreachable instance of an ambient actor. Besides this, the scenario will not be different to the previous step either.

Loss of connection at the arrival device Since the first step only produced the creation of the new ambient actor, getting loss of connection at this device will only imply the loss of this empty actor and the work of the ambient references.

6.4.1.3 While preparing the ambient actor contents to move

This step considers the reception and execution of the response of the arrival device. After this execution the new ambient actor at the other location starts to take the responsibilities of the *moved actor*. It starts to receive the messages sent by other actors over the network. However, it cannot process them until the next step, when it receives the last state of the moved object at the old location. Remember that in terms of network reconfiguration, at this moment two of the three mechanisms will start to work . These are the local message-rerouting and the updating by notification system for updating actors at third-party devices.

Loss of connection at the departure device Since the responsibility of continuing the process is at the departure device side (it has to send the last message) a disconnection of this device at this moment will be the most harmful. At the new location, it will have a empty ambient actor receiving the messages of the *moved actor* without any capacity of processing them. This disconnection also put out of action the updating by notification system. Actors at third-party devices that had not been updated before the disconnection of the departure device, will loose the communication with the moved actor. It means that these actor will not have access to the service provided by the moved actor.

Note that the solution to this problem is to use weak ambient references. As said before, these references will immediately look for new actors providing the same service (pattern) over the network.

Although semi-strong reference enables it to find the moved actor at a new location, it is not possible because the new actor representing the moved one, is still not *reachable* by the pattern-based lookup system. As explained in section 6.3.2.4 it will only happen once it gets filled and activated.

Loss of connection at the arrival device This case will also produce inconsistencies, since both actors found at the departure and third-party devices will continue sending messages that will never arrive to its destiny.

6.4.1.4 While restarting the ambient actor

This is the new state of the network. The new ambient object at the arrival device accomplishes with all the responsibilities of the moved actor, because it gets filled and activated in this step. The updating of the remote actor pointing to the old actor is done in such a way that local actors at this devices start to communicate locally with the moved actor.

Loss of connection at the departure device This disconnection does not mean anything from the point of view of actors at arrival device. For actors at third-party devices it will imply that they will not get updated by a notification from the old location, but in this case weak and semi-strong ambient references will be able to find the moved actor at the new location. Remember that since weak ambient references look for patterns, these are not forced to rejoin the same moved actor. By another hand a semi-strong ambient reference will join effectively the moved actor, as it matches with the pattern and the name this reference was looking for.

Loss of connection at the arrival device This last case of disconnection is quite similar to the first one. It means, the disconnection will be adequately handled by the three ambient references. The only difference will be that messages sent from third-party devices will be updated by notification. Note that this enables even the proper work of strong ambient reference, which gets updated at low-level for the network reconfiguration mechanism.

6.5 Evaluation of the AmbientTalk Mobility Model

The following evaluation of the model is done from two different points of view: The concepts that support this mobility model and its related works.

6.5.1 Decomposing the AmbientTalk Mobility Model

It has been said along this work that the AmbientTalk Mobility Model presented in this chapter was developed on top of the four concepts described in the chapters 2 to 5. These are mobile open networks, P2P network architecture, ambient-oriented programming and strong mobility. The following subsections explain in which ways these concepts are present in this model

6.5.1.1 The Model and the Mobile Open Networks

As explained in chapter 2, networks with mobile devices as participants have the same condition that the open networks. This is a dynamic reconfiguration over the time. In the case of the mobile networks, this is particularly due to the limitations of mobile devices such as scarce resources and connection volatility. The mobility of the actors residing in such devices must be aware to these change of conditions. It is complied by implementing the mobility model in a ambient-oriented programming language (AmbientTalk) that has mechanisms to deal with device mobility already.

6.5.1.2 The Model and the P2P Network Architecture

Chapter 3 described the properties p2p architectures such as decentralization, self-organization, fault-tolerance and scalability, matches properly to the requirements of mobile open networks. These architectures are already used in the AmbientTalk language by its pattern-lookup mechanism for acquaintance management between actors (such mechanism uses a broadcast P2P scheme). This mobility model also uses a P2P scheme as follows:

- Decentralized scheme for performing the actor mobility. This model does not require any central entity (like the director of MicroTapas or the Aglet-Host in Aglets) to move the actor from one location to another. It neither requires a centralized system for performing the network reconfigurability in any of the two scenarios described in chapter 6: when it is supported by the devices implied in the actor mobility and when it is supported by the ambient references.

- Fault-tolerant actor mobility. Disconnections of devices are supported by the ambient references at any moment during the move process.

- Self-organization mechanisms. There are two types of changes in the network that can occurs during the move process. The first one is mobility of the actor itself. The other is the potential disconnection of the devices that are performing the move. The first change is entirely contained in the actor mobility process implemented in this model. It implies the updating of the references to the moved actor in the departure, arrival and any other third-party devices, and the reconfiguration of the new actor to the new conditions found at the arrival device. The second change is responsibility of the ambient references pointing to the moved actor.

6.5.1.3 The Model and the Ambient-Oriented Programming

Two distinctive properties of ambient actors described in chapter 4 are used in the move process. Reified communications of actors are important for the success of strong mobility, as explained in the next point. The acquaintance management property (implemented by the pattern-based lookup mechanism) is used by the ambient references (which are actors as well) during the actor mobility. At a third-party device, an ambient reference pointing to the moved actor can take different decisions upon a disconnection of the departure or arrival device. This decision depends of the behavior each ambient reference implements. Chapter 6 described three types of ambient references. The difference is in the way these references use the pattern-based lookup mechanism.

6.5.1.4 The Model and the Strong Mobility

This concept allows actor to move without losing its availability for other actors over the network. The implementation of strong mobility in this model considers that an actor never lose its communication capacity. In this process the distinction between different communication states (provided by the communication reified mechanism) is useful to define the way in which the communication capacity of an actor will continue working during the move.

6.5.2 Mobility Model Comparisons

The following subsections compare the mobility model developed in this work with the closest approaches described in this dissertation. These are the three actor-based approaches created for open networks.

6.5.2.1 AmbientTalk and MicroTAPAS Mobility Model

- MicroTAPAS considers three types of mobility (user,terminal or mobile, and actor) whereas this model considers only two. Personal mobility which is related to the user session management was not included in the scope of this work.

- The MicroTAPAS mobility model considers two mobility entities that performs the mobility (mobility agent and manager). The actor mobility is controlled by the mobility managers of the system forming a hybrid P2P scheme in which they are the superpeers of the network. AmbientTalk mobility model uses a broadcast decentralized (broadcast) P2P system during all the move process.

- MicroTAPAS consider a instance of network reconfiguration mechanism after the arrival of the actor to the new location. This mechanism comprises mainly the reference updating to the this actor, the rebinding of resources (capabilities) and the reestablishment of actor relationships (role sessions). AmbientTalk Mobility Model does not provide mechanism to interact with physical resources. It just contains the reconfiguration message included in the moved method that is executed at first place upon arrival of the moved actor at the new location.

- MicroTAPAS does not have any abstraction to deal with device mobility. AmbientTalk mobility model includes the ambient references that already uses the mechanism implemented by this language for acquaintance management and reified communications.

- MicroTAPAS includes an abstraction for networks boundaries (domain and subdomains). It makes this distinction to delimitate the responsibilities of different directors and mobility managers in the network. AmbientTalk does not provide the notion of boundaries.

6.5.2.2 AmbientTalk and ChitChat Mobility Model

- Both ChitChat and AmbientTalk provide a lightly structured move process performed by three entities: the arrival and departure devices and the moved entity.

- Both languages allow programmers to build higher-level move methods. The move process occurs transparently for the programmers.

- Both languages implements strong mobility which means, moved entities never stop to work (partially though) during the move process.

- As MicroTAPAS, ChitChat does not provide any mechanism for interacting with the environment.

6.5.2.3 AmbientTalk and SALSA Mobility Model

- SALSA move process is initiated by a asynchronous message that allows continuations which will be execute at the new location of the actor. AmbientTalk occurs the same case with the higher-level move method and the reconfiguration message.

- In SALSA programmers interact directly with the universal naming service protocol (UNAP) which means they have to know somehow the ual or uan of the actors implied in the move process. AmbientTalk implements a pattern-based lookup mechanism. A pattern is much more like a service provided by an actor.

- SALSA consider a network reconfiguration using a DHT-based P2P architecture. In addition it provides abstractions for representing the internal resources of devices. AmbientTalk provide a decentralized P2P scheme for network reconfiguration but, as said above, it does not provide abstractions for interacting with resources directly.

6.6 Conclusion

This chapter has presented the proposal of this thesis. The AmbientTalk mobility model is based partially in abstractions already built for other contexts such as the ambient references. This model was the result of the application of four concepts described in previous chapter, and the properties of previous mobility models also described in this work. The final conclusion will summarize this mobility model (chapter 8). The following chapter will apply this model to a concrete type of application belonging to the AmI field. It will be the validation of this model.

Chapter 7

Application and Validation

7.1 Introduction

This chapter presents an application for the AmbientTalk mobility model seen in the previous chapter. In summary, this model considers the following:

- Support for device mobility by using *Ambient References*, an abstraction at the programming language built on top of the ambient-oriented model of AmbientTalk (reified communications and acquaintance management mechanisms) described in chapter 4.

- Support for strong mobility of ambient actors by using the `moveActor` primitive. It can be included in higher order *move* methods created by an AmbientTalk programmer.

The application selected for the validation of this model is called *Follow-Me*, one of the most representative type of applications of the new field in distributed computing known as *Ambient Intelligence* [ISTAG, 2003]. As it is briefly explained in the next section, applications developed for this field run over a mobile (open) network. Thus, the validation of the mobility model will consist in the implementation of a software design pattern for the *Follow-Me* applications.

Besides the implementation of the *Follow-Me* pattern, this case will also validate the concepts identified along this work to develop applications in mobile networks. These are P2P architectures, ambient actors and strong mobility.

7.2 Ambient Intelligence

Ambient Intelligence (acronym AmI) is a term adopted by the European Council's IST Advisory Group [ISTAG, 2003] to refer to the vision that technology will become invisibly embedded in the environment of the people, wherever they are, adaptive to them and their contexts, and enabled by simple and effortless interactions[1]. This vision coincides with the concept of *Ubiquitous Computing* described by Mark Weiser in [Weiser, 1991].

In terms of infrastructure, these properties suggest a network with embedded devices representing the available environment and small mobile devices for interacting with such environment. These mobile devices could not be required in case of embedded technology with perceptual capacity for the interaction with users (using speech or vision [MIT-LCS, 2004]). But these mobile devices can become the environment itself, like in the case of PANs described in chapter 2. Whatever road is taken, it will correspond to the description of the mobile (open) network given in this work (chapter 2).

The AmI vision mentioned above also proposes an adaptive system available to the users anywhere, in any context. The application described in the following section will demonstrate that the AmbientTalk mobility model provides proper abstractions for developing such a type of systems.

7.3 *Follow-Me* Applications

The *Follow-Me* application is one of the most studied cases in the field of Ambient Intelligence [Landay, 2003, Berger et al., 2003, Satoh, 2002, Priyantha et al., 2000]. The main idea is to provide the user with an application (a service) that follows him, adapting itself to the different contexts found at the different places where the user moves around. The following subsection will exemplify this application with a story. Posteriorly, the implementation of this application will be described.

7.3.1 The Story: Bob's Mobile Desktop

One story for explaining this scenario could be the following:

> *Bob is writing a document on the PC at home. While he writes, he is listening to some music and chatting with a few friends. Suddenly, his daughter Alice enters the studio where Bob is and asks him whether she can use the PC to do her homeworks. Gently, Bob leaves the studio in order to take his laptop to his bedroom and continue working there.*

7.3.1.1 The Problem

The problem in this story is that Bob needs somehow to reproduce on his laptop at his bedroom, the same work conditions he had on the PC at the studio. In this case, he needs the text editor with his document, the music player with the song he was listening and his chat session. To achieve this is not the issue, but the way in which these conditions are recovered. The solution to this problem implies to move all those services required to enable Bob to work properly on his laptop.

[1]These and other properties of the AmI vision are mentioned in [Lindwer et al., 2003].

7.3.1.2 The Solution

The AmbientTalk mobility model enables this story to have the following epilog:

> *Bob leaves the studio without doing anything on the PC, goes to his bedroom, opens the laptop (starts it if required) and after a while, the applications he was using on the PC appear on the laptop's desktop (text editor, music player and chat). He realizes that while he was changing of work place, some friends continued chatting with him. Thus, Bob answers them.*

The AmbientTalk mobility model implements strong mobility of actor-based applications. As such, the actors can receive messages even while they are moving to other location (remember that actors can can not process messages while they move). That is why Bob can receive messages in his chat session from its friends even during his move.

Note that this story could have a different (but less convenient) result for each type of mobility described in the chapter 5:

- If weak mobility would have been used (dead code traveling through the network), Bob would have to store his work and music somewhere on the network, and after restarting the applications at the new location he would have to download the files.
- In case of implementing the *Follow-Me* application using semi-strong mobility (control context must be manually converted to data, which is then moved through mobility. Afterwards the programmer must manually recover the control context from the data) at the new location but the control context is converted into data (serialized) when it is moved), Bob would get the same final result of the strong mobility case, but he would have lost his chat session during the move of the chat application.
- The full mobility is hard to imagine in this case (data, control and resource contexts moved to the new location). It could be something like the system trying to install on the laptop, the driver of the audio card found on the PC.

The next section explains the implementation in AmbientTalk of the solution for this application.

7.3.2 Implementation of a *Follow-Me* Application

The implementation of the *Follow Me* application considers the following systems working together:

1. A mechanism that determines the closest computer to the user (Bob). This could be accomplished by demanding the user to log in each time he arrives

to a computer. Actually, there are also location-support infrastructures that comply with this requirement without explicit interaction of the user. These infrastructures will be briefly described later on.

2. A mechanism that move the applications to the user once he is localized at another computer. This work proposes the development of a *Follow-Me* pattern for this purpose.

7.3.2.1 Location-Support Infrastructure

Actually, there are several works oriented to develop location-support systems [Priyantha et al., 2000, Ward, 1998]. These systems are like indoor Global Position Systems (GPS) that consist of mobile devices capable to interchange information which is embedded in the space (a room, a building, etc.), in order to determine its current location. Based on that information, these systems enable users (and their mobile devices) to interact with their *closest* environment.

The scenario described in the previous section assumes the existence of a location-support infrastructure in Bob' house. As said before, it mainly will avoid Bob from having to log on each computer he is visiting.

AmbientTalk and Location-Support Systems Location-support systems suggest a new differentiation criteria between entities offering similar services, that could be considered somehow in the AmbientTalk acquaintance management mechanism. In the current version of AmbientTalk, this mechanism looks for actors providing services over the entire network. The question is how can an actor requiring a service discover the *closest* actor providing that service. There is no way until the moment in this mechanism to focus the searching over a specific segment of the network[2], like for instance the *closest* one. Perhaps it implies to consider the notion of *boundaries* which can be intentionally defined, like the ones described in [Cardelli, 1999].

It is something that can also be related to the notion of *overlay networks* explained in chapter 2. Note that a similar scenario occurs when a mobile device is joining more than one network at time (e.g. a device simultaneously connected to a WLAN and a PAN). The decision of looking for a service in one or another network (or both) could be taken by the user.

In the context of this work it will be assumed that this criterium is already implemented in the acquaintance management mechanism. It means, actors can discover the closest other ones that provide the services they are requiring.

[2]There is a primitive way in which this can be done: actors could store information about their location. Then, an actor which searches for a certain pattern joins with a number of actors providing this pattern. The actor could then manually scan the list of joined actors and filter out the *closest* one based on the location information he gets from each actor. However, a nicer solution could be achieved by the inclusion of boundaries to the model

7.3.3 The *Follow-Me* Pattern

This pattern ensures that the actor-based applications follow their users whenever they move over the network. They will remain available depending of the adaptability of these applications to the new contexts found at new locations. This pattern focuses mainly on the mobility of the actor-based applications, rather than their adaptability.

The *Follow-Me* pattern assumes two applicability conditions:

1. The use of mobile actor-based applications, it means, applications composed by actors that can be moved over the network.

2. A location-support system deployed over the space in which users move around. It implies mobile and embedded devices supporting such a system.

7.3.3.1 Participants

This pattern requires four type of actors: `FollowMe`, `User`, `AmbientApplication` and `AmbientSession` actors.

The `FollowMe` actor This type of actor resides on the work stations[3] (like the Bob's PC and laptop). `FollowMe` actors are responsible to keep users and their sessions together, looking for such sessions over the network and moving them if required.

The `User` ambient reference This ambient reference resides on the mobile devices carried by the users. It informs the user identification to the `FollowMe` actor residing on the computer in which the user is currently working.

The `AmbientSession` actor This actor is the container of all applications opened on a computer by a user to do his work. This session moves together with the applications, in the same spirit of the *swarm* object described in [De Meuter, 2004]. It will be explained in the next section.

The `AmbientApplication` actor This actor represents a application with the capability to move and adapt itself to the conditions of the new locations.

7.3.3.2 Collaborations

The *Follow-Me* pattern works as follows:

1. The first time that a `UserRef` ambient reference matches a `FollowMe` actor, it sends a message to that actor asking it for the creation of a user session. Thus, the `FollowMe` actor creates an `AmbientSession` actor for this user.

[3]`FollowMe` actors are no limited to be on work stations, but for understandability of the pattern we will assume this. Different scenarios will be discussed in section 7.3.4

2. The user opens its required applications (`AmbientApplication` ones) and registers them in the `AmbientSession` (which may be automatically performed).

3. The user leaves the current location and the `UserRef` reference starts immediately to look for another `FollowMe` actor. At this moment the `AmbientSession` actor is already prepared to be moved.

4. The user arrives to a new location. The `UserRef` ambient reference finds the new `FollowMe` actor. Then, the reference informs the user identification to this actor and it starts the search for its `AmbientSession` over the network.

5. Once the `FollowMe` actor finds the required `AmbientSession` actor, the former send a message to the latter asking it to move to the new location (computer) where the user is working.

6. Subsequently, the `AmbientSession` actor sends messages to its containing `AmbientApplication` actors demanding them to move.

7. Finally, the `AmbientApplication` actors resume their full activities (if possible, depending of their reconfiguration results).

7.3.3.3 Implementation

The full implementation of the *Follow-Me* pattern can be found in the appendix A. This section will highlight some relevant parts of the code for the validation of this work.

In few words, the implementation of the *Follow-Me* pattern consist of *finding the user's session over the network* and *moving it to the new user's location*. As it is shown in the following sections, the first part is accomplished by the use of ambient references, whereas the second part is fulfilled by the use of `moveActor` that allow strong mobility of the actors.

Ambient References The ambient references are used in the searchings required by this pattern. The first one is performed by the `UserRef`, which is itself an ambient reference. It looks for a ”followme” pattern, certainly provided by the `FollowMe` actors (see the code below). The implementation of this ambient reference is similar to the implementation of the weak ambient reference described in chapter 6. The difference is that the former handle the information of the user (in this implementation it corresponds just to an extra field with the user name).

The second lookup is performed by the `FollowMe` actor in order to find the `AmbientSession`. The extract of code shows the `findSession` method that is called by the `UserRef` reference correspondent to the user that is currently in the computer (where the `FollowMe` resides). The session field is filled with a weak ambient reference. The next line invokes the `moveWhere` method which is the higher-order move method implemented by the `AmbientSession` actor. Note that since the `itsSession` field contains an ambient reference (that is an actor, as explained in chapter 6), the user could eventually starts to interact with this session.

```
followMeBehaviour :: object({
      ...
      itsPattern: "followme";
      init() :: { publish(itsPattern) };
      ...
      findSession(aUserName) :: {
            itsUserName:= aUserName;
            itsSession:=
                WeakAmbientRef(itsUserName);
            itsSession#moveWhere(thisActor(),
                                 itsResources);
            itsSession};
      ...
      getSession(aUserName)::
                {if(aUserName=itsUserName,
                    itsSession)};
      ...
   })
```

The Higher-Order Move Method As previously mentioned, `AmbientSession` and `AmbientApplication` actors move together to the new location. The way to do it is similar to the one described in the *swarm* pattern [De Meuter, 2004]. The code below shows that the invocation of the `moveWhere` method of the session, implies the invocation of the `moveWhere` method of the applications. This mechanism is a bit more restricted than the *swarm* pattern, since in this case the move methods of the applications must have the same name. After the iteration through the applications contained in the session, it calls the `moveActor` primitive in order to move itself to the new location.

```
ambientSessionBehavior:object({
    ...
    itsApps:vector.new();
    ...
    init():: {publish(itsUserName)};
    ...
    moveWhere(aFollowMe,aResources):: {
        itsApps.iterate(el.moveWhere(aFollowMe,
                                     aResources));
        moveActor(aFollowMe,void)};
    ...
})
```

The implementation of the `moveWhere` method of an application includes the invocation of the `moveActor` primitive that enable the application actors to move as well. In this case this primitive is invoked with a message as a second argument that calls the `resume` method of the application. It was explained in chapter 6 that this second argument corresponds to the first message to be processed by the actor at the new location. The purpose of the `resume` method in this case is to include in its body all needed to adapt the application to the new context.

```
ambientAppBehavior:object({
    moveWhere(aFollowMe,aResources):: {
        moveActor(aFollowMe,
              createMessage(thisActor(),
                    thisActor(),
                    "resume",
                    [aResources]))};
    resume(aResources):: { ... };
});
```

7.3.4 Related Scenarios

The following scenarios can occur considering the conditions provided by the location-support system and the implementation of *Follow-Me* pattern:

Two users arriving to the same location There are no restrictions for avoiding two or more people to have their sessions on the same machine at the same time. A way to limit this (if required) could be to make the user invariable in the `FollowMe` actors.

`FollowMe` actors located in embedded or mobile devices This scenario is quite the spirit of the ambient intelligence. It stresses the property of adaptability of the applications.

`AmbientSession` actors moved to mobile devices It can be an alternative way to move applications through the network. Since this is a strong mobility, users could keep working on that applications on the device, even without network connection.

7.4 Conclusion

This chapter has presented the *Follow-me* applications as a way to validate the AmbientTalk mobility model proposed in this work. These applications correspond to the field of Ambient Intelligence, which, as explained in this chapter, commonly suppose dynamically reconfigurable environments, like the one found in open networks.

New infrastructures are being developed to support AmI applications. Such is the case of location-support systems that deal with the physical discovery of the participants of a network. Several works can be identified in terms of the software required for the context of AmI. AmbientTalk is one of them. The pattern proposed in this chapter to build location-aware applications such as the *Follow-Me* ones, takes advantage of the mechanisms found in AmbientTalk to deal with devices and actor mobility.

Chapter 8

Conclusions

This dissertation has described the AmbientTalk mobility model developed in this thesis to deal with both device and ambient actor mobility. While the mechanisms for supporting device mobility was already included in the AmbientTalk programming language, the ambient actors mobility were the result of the application of the concepts such as open networks, p2p network architectures, ambient actors and strong mobility. The following two sections summarize the Ambient mobility model and relate it with the concepts mentioned before. Subsequently, the applicability of this model will be discussed, together with directions for future work.

8.1 Problem Statements (revisited)

The following are the problems identified in the introduction of this dissertation, and the solution found in this work.

8.1.1 Dealing with Mobile Networks

Mobile networks have a dynamic reconfiguration over the time due to the limitations of their participants (devices). In this proposal, the problems produced by this property of the open networks were tackled with the mechanisms already present in the AmbientTalk programming language (non-blocking communication primitives, reified communication traces and ambient acquaintance management described in section 4.7). Such mechanisms have been combined with the actor mobility in the following scenarios:

- To avoid that mobility of devices affects the actor move process (section 6.4). It was explained in this section that mixing both mobility the most of the crashes during the actor move process can be avoided.

- To implement the *Follow-Me* application (section 7.3). This application required services (Bob's desktop) to hop from one computer to another. This application also required a location-support systems based on mobile and embedded devices.

The conclusion in this case is that for the purpose of ensuring (as much as possible) the availability of the services provided over the mobile network, the combined solutions of mechanisms for mobile and actor devices get better results than the independent solutions provided by the mechanisms of one or another.

8.1.2 Previous Code mobility Issues

The problems mentioned in the introduction respect to previous code mobility implementations, pointed to complexity (in case of Java) and the lack of mechanisms for interacting with dynamically reconfigurable networks (in case of agent technology). The implementation of this model care about both issues as follows:

- Complexity for working with actor mobility was avoiding by implementing a lightly structure move process which consist of a primitive that can be included in any higher-level move method developed at AmbientTalk programming level (see section 6.3.1). The move process is totally transparent for the user. He does not have to take care about distribution and concurrency problems because these are already tackled at AmbientTalk implementation level (by implementing the ambient actor model described in section 4.7.1.1). He neither has to take car about the network reconfiguration after the move because it is already included in the implementation of the AmbientTalk mobility model (section 6.3.3)

- Respect to the second issue described above, it is solved by the ambient-oriented programming paradigm implemented in AmbientTalk (section 4.7). Its implementation considers a set of abstractions developed to discover services over the network (pattern-based lookup, section 4.7.1.1) and to avoid inconsistency states in the systems due to connection volatility of the mobile devices (communication states, section 4.7.1.1).

  AmbientTalk programmers can interact directly with these mechanisms by means of a small set reflective methods. But these mechanism can be encapsuled in a higher-level abstraction. The AmbientTalk mobility model includes ambient references that are abstractions to these mechanisms (section 6.2.1).

8.2 Approach (revisited)

As explained in the introduction of this dissertation consists of a set of combined mechanisms: ambient references abstracting the AmbientTalk mechanisms for dealing with mobile devices, the strong mobility of actors, and the network reconfiguration mechanism performed after the actor's move.

Some design considerations to this model are the following:

- This mobility model extends the P2P network scheme implemented in AmbientTalk:
    - None central entities are required for executing the actor mobility and network reconfigurability.
    - Actor mobility is a fault-tolerant mechanism since disconnections of devices are supported by the ambient references at any moment during the move process.
    - This model provides a structure that can organize itself after the actor's move and after a unexpected disconnection of the devices that are performing the move. This second case is responsibility of the ambient references pointing to the moved actor found at any device over the network.
- As mentioned before, this model is strongly supported by the mechanisms implemented by AmbientTalk for providing communication states and pattern-based lookup mechanism. While the former is used to allows actor to move without losing its availability for other actors over the network, the latter is used by the ambient references during the actor mobility.

8.3 AmbientTalk Mobility Model in a Nutshell

The AmbientTalk mobility model proposed in this thesis works as follows:

- This model proposes to deal with device and actor mobility by using two abstractions at AmbientTalk programming language level: the ambient references and the move primitive.
    - The ambient references are an abstraction of the mechanisms AmbientTalk provides to deal with device mobility, namely reified communication and acquaintance management.
    - The `moveActor` primitive is the abstraction of the actor strong mobility mechanism implemented in this work. It receives as parameters a remote actor residing in the future new location of the actor that needs to be moved, and a message that will be used for coordination (adaptation) purposes. It will be the first message to be executed once the moved actor starts to process messages at the new location. This primitive can be used in any method of an ambient-actor, which can build more high level abstractions on top of it.
- The actor move process is a shared responsibility between the current (departure) device of the actor and its future (arrival) device.

- This process is based on four communications. The first one corresponds to the invocation of the `moveActor` primitive and the three last ones correspond to virtual machine-level commands sent between the devices implied in the move.

- The sequence of steps corresponds to the execution of such communications. The steps are the following:

  1. An actor is demanded to move by receiving a move method (a method with the `moveActor` primitive inside its body).
  2. The execution of that primitive produces that device starts the process by checking low-level preconditions for the move. If these are properly fulfilled the departure device sends the `moveActor` command to the arrival device.
  3. The execution of this command at the arrival device creates a deactivated and empty actor (without processing capacity, behavior, state, and mailboxes' contents) but ready to receive messages from other actors. Then the arrival device sends back a `resultMoveActor` command. This command passes as parameter the reference to the new actor.
  4. The execution of this command at the departure device produces the delegation of the communication capacity of the moved actor to the created actor at the arrival device. At this moment the departure device stops the processing capacity (thread) of the moved actor and sends the `moveActorContents` command. It has as parameters the behavior (with the state) and mailboxes' contents of the moved actor.
  5. The execution of this last command at the arrival device loads the contents in the new actor and starts its thread.

- The network reconfiguration mechanism of this model will update the references at third-party devices (neither the departure, nor the arrival device) pointing to the moved actor at its old location. This way chained forwarding (rerouting) of messages is avoided.

8.4 Applications for Mobile Open Networks

Chapter 7 considered the implementation of a AmI application called *Follow-Me* for validating the AmbientTalk mobility model. A *Follow-Me* application follows its user around a space (a house, a building) covered with embedded devices that provide information about the location of people and resources. The goal was to create a pattern for developing such a type of applications using the mobility model proposed in this work. The implementation consisted of finding the user's session over the network and moving it to the new user's location. The result of this experiment demonstrated that both actions are properly accomplished by the model (the first one using ambient references and the second one using strong mobility of actors). The assumption was to consider the availability of the location-support infrastructure for this exercise and the adequate communication between such a system and AmbientTalk. Although it is possible to simulate the detection of the user mobility at software level (for instance, by sending messages directly to the `FollowMe` actor to simulate disconnection) the utilization of this pattern with real ubiquitous infrastructure remains future work.

It is to be said that the communication with such a infrastructure would be rather a matter of the internal implementation of AmbientTalk than an issue at the programming level. Therefore, the *Follow-Me* would not change. What effectively could be added at the programming level is an abstraction for representing the information about physical location of mobile devices.

8.5 Future Work

The following points correspond to possible continuation of the work presented in this dissertation.

- As it was mentioned in this work, strong mobility of actors implies to adapt themselves to the conditions of the new locations. This work has focused on the move process itself more than the adaptability mechanism. It could be necessary for improving the applicability of this model.
- The security is an issue in this work. There are no mechanisms to limit the capacity an actor has to modify the device where it is residing. It can be worst in the context of this work, in which actors are enabled to hop from one device to another.
- The strong mobility mechanism requires the notion of distributed garbage collection described in [Van Custem and Mostinckx, 2004] since it creates new actors each time an actor moves, but it does not remove the old ones, even if these are not being used anymore.
- The previous section mentioned the necessity of testing AmI applications using new hardware infrastructure. It could be interesting to know what the implications of these systems are for software development.

Appendix A

AmbientTalk Applications

The following applications were developed in the context of this work for validation purposes. These are coded in AmbientTalk. A guide of the semantics used in this programming language can be found in [Dedecker, 2005b]. The two applications described in this appendix are:

- Applications for testing the strong mobility of ambient actors, and the network reconfiguration that takes place after the move.
- The implementation of the actors required by the *Follow Me* pattern.

A.1 Applications for Testing the AmbientTalk Mobility Model

The programs presented in this section were developed to test the AmbientTalk mobility model (explained in chapter 6). Each section corresponds to a test created for the three different devices identified in the explanation of the mobility model. These are the departure device, the arrival one, and any other third-party device.

These test applications use an object that represents the device. It contains in its implementation a `displayActor (actor)` method, that shows in which device are the actors.

```
device:: object({
  theName: void;
  cloning.new(aName)::theName:=aName;
  getName()::theName;
  setName(aName)::theName:=aName;
  displayActor(actor)::display(actor, " is in ",
                               theName, eoln)
});
```

A.1.1 Actor Mobility Implementation at the Departure Device

This device is the current location of the actor to be moved (called `locA`). The following code contains an implementation of a *move* method (called `moveWhere`) which contains the `moveActor` primitive defined in chapter 6.

The last line of this code (`locA#displayMe()`) will be executed after the move of the actor in order to test that the communications to the moved actor from this devices will not be affected.

```
{ thisDevice: device.new("Device-1");

o:object({
    theName: void;
    cloning.new(aName)::theName:=aName;
    init():: publish("test1");
    displayMe():: thisDevice.displayActor(theName);
    moveWhere(aRef):: when(aRef#getProvider(),
                moveActor(content,
                          createMessage(thisActor(),
                                        content,
                                        "displayMe",
                                        [])))

}).futuresMixin();

locA: actor(o.new("Actor-A"));

startNetwork();

locA#displayMe() }
```

A.1.2 Actors at the Arrival Device

This device represents the new location of the moved actor. The following code creates a new actor at this device (`locB`) and an ambient reference (concept explained in chapter 6) to the actor found at departure device (`remA`) that will be moved to this device. The idea is to test the communications with the moved actor will not be affected after the move.

```
{ thisDevice: device.new("Device-2");

o: object({
    theName: void;
    cloning.new(aName)::theName:=aName;
    init()::publish("test2");
    displayMe():: thisDevice.displayActor(theName)
}); locB: actor(o.new("Actor-B"));

startNetwork();
locB#displayMe() }

{ remA: ambientRef("test1"); remA#displayMe() }
```

A.1.3 Actors at a third-party Device

This device represents any device that is neither the departure device, nor the arrival one of the moved actor. Two ambient references are created at this device: `remA` is a reference to the moved actor. The `remB` is a reference to the actor created at the arrival device. This code shows that the actor represented by `remA` is demanded to move at the location of `remB` (the line is `remA#moveWhere(remB)`). Note that the last line of this code (`remA#displayMe()`) is called after the actor was moved. The purpose of this is to test the communications with the moved actor will not be affected after the move.

```
{ remA: ambientRef("test1");
  remB: ambientRef("test2");
  startNetwork();
  remA#displayMe();
  remB#displayMe() }

{ remA#moveWhere(remB);
  remA#displayMe() }
```

A.2 *Follow Me* Pattern Implementation

The following code corresponds to the implementation of the actors required by the *Follow Me* pattern described in chapter 7. At the end of this section are also included some examples of mobile applications.

A.2.1 `FollowMe` Actor

This is the actor that calls the `AmbientSession` actor to move. It happens only in the case of a `UserRef` ambient reference invokes the `findSession(aUserName)` method. Note that this last method uses a weak ambient reference (explained in the chapter 6) to find the `AmbientSession` of the user over the network.

```
followMeBehaviour :: object({
      itsSession: void;
      itsUserName: void
      itsResources: void;
      itsPattern: "followme";
      cloning.new(aResources) :: {
            itsResources:= aResources};
      init() :: { publish(itsPattern) };
      findSession(aUserName) :: {
            itsUserName:= aUserName;
            itsSession:=
                WeakAmbientRef(itsUserName);
            itsSession#moveWhere(thisActor(),
                                 itsResources);
            itsSession};
      getSession(aUserName)::
                     {if(aUserName=itsUserName,
                         itsSession)};
      createSession(aUserName):: {
            itsUserName:= aUserName;
            itsSession:=
                AmbientSession.new(itsUserName,
                                   itsResources);
            itsSession}
   })

FollowMe(aResources) ::
          actor(followMeBehaviour.new(aResources))
```

A.2.2 `User` Ambient Reference

This is an ambient reference that resides on the mobile device of the user. Once it finds a `Follow Me` actor, it either sends a `createSession` message or a `findSession` in case of the AmbientSession was previously created already. This part of the behavior is found in the `joined(aResolution)` method of this ambient reference.

The implementation of the `UserRef` is based on the implementation of the weak ambient references (`WeakAmbientRef`). Such a type of ambient reference was already implemented in AmbientTalk [Dedecker, 2005a] at the moment of the creation of this pattern.

```
userRefBehaviour :: object({
      itsName : void
      itsPattern : void;
      itsRef : void;
      itsInitiated : false;
      cloning.new(aPattern,aName)::
```

```
                     {itsPattern:= aPattern ;
                      itsName:= aName };
init() :: { add("required",itsPattern) };

joined(aResolution) :: {

    if(is_void(itsRef), {
       itsRef := provider(aResolution);
       if(itsInitiated,
          { outbox.add(createMessage(
                               thisActor(),
                               itsRef,
                               "findSession",
                               [itsName]))
          }, {
            itsInitiated:=true;
            outbox.add(createMessage(
                               thisActor(),
                               itsRef,
                               "createSession",
                               [itsName]))
          });
       toForward: inbox.asVector();
       toForward.iterate({
          if(not(containsBehaviour(el.getName())),
            { outbox.add(setMsgTarget(el.copy(),
                                      itsRef));
              inbox.delete(el) })})
    })
};

disjoined(aResolution) :: {

    if(provider(aResolution) ~ itsRef, {
        itsRef := void;
        outbound : outbox.asVector();
        outbound.iterate({
            aMsgTarget : el.getTarget();
            if (aMsgTarget ~ provider(aResolution),
                { outbox.delete(el);
                  inbox.add(el)})
        });

        if (joinBox.length() > 0,
                itsRef := provider(
                            joinBox.asVector().get(1)))
        });
        delete("disjoined",aResolution)

    };

    in(aMsg) :: {
       if(not(is_void(itsRef)) &
              not(containsBehaviour(aMsg.getName())),
                  {outbox.add(setMsgTarget(
                                        copyMsg(aMsg),
                                        itsRef));
                   inbox.delete(aMsg)})
    }

 })

 UserRef(aName) ::
    actor(UserRefBehaviour.new("followme",aName))
```

A.2.3 **`AmbientSession`** Actor

This is the actor that will be moved as well as the `AmbientApplication` actors it contains. The `moveWhere(aFollowMe,aResources)` method is the responsible of the move of this actor and the `AmbientApplication` actors.

```
ambientSessionBehavior:object({
    itsUserName:void;
    itsApps:vector.new();
    cloning.new(aUserName):: {
        itsUserName:=aUserName};
    init():: {publish(itsUserName)};
    moveWhere(aFollowMe,aResources):: {
        itsApps.iterate(el.moveWhere(aFollowMe,
                                     aResources));
        moveActor(aFollowMe,void)};
    setApplication(anApp):: itsApps.add(anApp);
})

AmbientSession(aUserName) ::
      actor(ambientSessionBehaviour.new(aUserName))
```

A.2.4 `AmbientApplication` Actor

This actor contains the behavior of a mobile application. Thus, the applications that will be considered in the *Follow Me* mechanism, should extend this behavior. Note that the `resume` method is the place in which the application can reconfigurate itself to the resources found in the new device.

```
ambientAppBehavior:object({
    moveWhere(aFollowMe,aResources):: {
        moveActor(aFollowMe,
              createMessage(thisActor(),
                    thisActor(),
                    "resume",
                    [aResources]))};
    resume(aResources):: {'It reconfigures itself
                  upon arrival and restarts.'};
});

ambientApplication() ::
                actor(ambientAppBehaviour.new())
```

Mobile Text Editor A very simple (and toy) version of the text editor actor could be the following:

```
textEditorBehavior:ambientAppBehavior({
    itsText: void;
    write(aNewText):: {itsText:= itsText + aNewText};
    displayText():: display(itsText);
    resume(aResources):: {display("Text Editor ready to work!");
                  displayText()}
})

TextEditor():: actor(textEditorBehavior.new())
```

Mobile Music Player An also simple version of the music player actor could be the following:

```
musicPlayerBehavior:ambientAppBehavior({
    itsSong: void;
    addSong(aSong):: itsSong:= aSong;
    play():: display("Playing the song: ", itsSong);
    resume(aResources):: {display("Music Player ready to work!");
                  play()}
})
```

```
MusicPlayer():: actor(musicPlayerBehavior.new())
```

Bibliography

[Agha, 1986] Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. MIT Press.

[Baude et al., 2003] Baude, F., Caromel, D., and Morel, M. (2003). From dstributed objects to hierarchical grid components. In *On The Move to Meaningful Internet Systems 2003: Coopis, DOA, and ODBASE, volume 2888 of Lecture Notes in Computer Science*, pages 1226–1242.

[Berger et al., 2003] Berger, S., Schulzrinne, H., Sidiroglou, S., and Wu, X. (2003). Ubiquitous computing using sip. In *NOSSDAV 03*. ACM.

[Bernard Traversat and Pouyoul, 2003] Bernard Traversat, M. A. and Pouyoul, E. (2003). Project jxta: A loosely-consistent dht rendezvous walker.

[Briot and de Ratuld, 1988] Briot, J.-P. and de Ratuld, J. (1988). Design of a distributed implementation of abcl/i. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 15–17. ACM Press.

[Briot et al., 1998] Briot, J.-P., Guerraoui, R., and Lohr, K.-P. (1998). Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329.

[Callsen and Agha, 1994] Callsen, C. and Agha, G. (1994). Open heterogeneous computing in actorspace. *Journal of Parallel and Distributed Computing*, pages 289–300.

[Cardelli, 1995] Cardelli, L. (1995). Obliq: A language with distributed scope. *22nd Annual ACM Symposium on Principles of Programming Languages*, pages 286–297.

[Cardelli, 1999] Cardelli, L. (1999). Abstractions for mobile computation. In *Secure Internet Programming*, pages 51–94.

[Carton and Mesaros, 2004] Carton, B. and Mesaros, V. (2004). Improving the scalability of logarithmic-degree dht-based peer-to-peer networks. *Euro-Par 2004*.

[Cerulean Studios, LLC, 2005] Cerulean Studios, LLC (2005).

[Clements et al., 1997] Clements, P. E., Papaioannou, T., and Edwards, J. (1997). Aglets: Enabling the virtual enterprise. In *Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement*.

[Cohen, 2003] Cohen, B. (2003). Incentives build robustness in bittorrent.

[De Meuter, 2004] De Meuter, W. (2004). *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel.

[De Meuter et al., 2004] De Meuter, W., D'Hondt, T., and Dedecker, J. (2004). Pico: Scheme for mere mortals. In *Proceedings of the First International Lisp Workshop*.

[Dedecker, 2005a] Dedecker, J. (2005a). Ambient-oriented programming.

[Dedecker, 2005b] Dedecker, J. (2005b). AmbientTalk: The programming language kernel for ambient computing.

[Ding and Bhargava, 2003] Ding, G. and Bhargava, B. (2003). Peer-to-peer file-sharing over mobile ad hoc networks.

[Eberspcher et al., 2004] Eberspcher, J., Schollmeier, R., Zols, S., and Kunzmann, G. (2004). Structure p2p networks in mobile and fixed environments.

[Emir Halepovic, 2003] Emir Halepovic, R. D. (2003). Jxta performance study.

[Fugetta et al.,] Fugetta, A., Picco, G., and Vigna, G. Understanding code mobility. In *IEEE Transactions on Software Engineering, 24(5)*, pages 342–461.

[Gelernter, 1985] Gelernter, D. (1985). Generative communication in Linda. In *ACM Transactions on Programming Languages and Systems 7, 1*.

[Ion Stoica and Balakrishnan, 2001] Ion Stoica, Robert Morris, D. K. F. K. and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications.

[Islam and Fayad, 2003] Islam, N. and Fayad, M. (2003). Towards ubiquitous acceptance of ubiquitous computing.

[ISTAG, 2003] ISTAG (2003). Ambient intelligence: From vision to reallity.

[J. Kurhinen and Vuori, 2004] J. Kurhinen, M. Vapa, M. W. N. K. and Vuori, J. (2004). Short range wireless p2p for co-operative learning.

[J. Liang and Ross, 2004] J. Liang, R. K. and Ross, K. (2004). Understanding kazaa.

[Jul et al., 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*.

[Krco et al., 2005] Krco, S., Cleary, D., and Parker, D. (2005). P2P mobile sensor networks. In *38th Hawaii International Conference on System Sciences*.

[Landay, 2003] Landay, J. A. (2003). Design patterns for ubiquitous computing. *IEEE computer ubicomp*.

[Lange and Oshima, 1998] Lange, D. B. and Oshima, M. (1998). Programming and deploying java mobile agents with aglets. *Addison Wesley*.

[Lea, 1999] Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition. Online Supplement at `http://gee.cs.oswego.edu/dl/cpj`.

[Leopold et al., 2003] Leopold, M., Dydensborg, M. B., and Bonnet, P. (2003). Bluetooth and sensor networks: A reality check. In *First International Conference on Embedded Networked Sensor Systems*.

[Li, 2003] Li, S. (2003). Jxta 2: A high-performance, massively scalable p2p network.

[Lin and Chlamtac, 2001] Lin, Y.-B. and Chlamtac, I. (2001). *Wireless and Mobile Network Achitectures*. John Wiley and Sons.

[Lindwer et al., 2003] Lindwer, M., Marculescu, D., Basten, T., Zimmerman, R., Marculescu, R., Jung, S., and Cantatore, E. (2003). Ambient intelligence visions and achievements: Linking abstract ideas to real-world concepts.

[Liskov, 1988] Liskov, B. (1988). Distributed programming in argus. In *Communications Of The ACM, 31(3)*, pages 300–312.

[Luhr, 2004] Luhr, E. (2004). Mobility support for wireless devices - within the TAPAS platform. Master's thesis, Norwegian University of Science and Technology.

[Mascolo et al., 2002] Mascolo, C., Capra, L., and Emmerich, W. (2002). Mobile computing middleware. *Advances Lectures on Networking*.

[Mathieu Jan, 2004] Mathieu Jan, D. A. N. (2004). Performance evaluation of jxta communication layers.

[Miller, 2000] Miller, M. (2000). The e programming language, the secure distributed pure-object platform and p2p scripting language for writing capability-based smart contracts.

[Milojicic et al., 2000] Milojicic, D., Douglis, F., and Wheeler, R. (2000). *Mobility: Processes, Computers, and Agents*. Addison-Wesley, second edition.

[Minar et al., 2001] Minar, N., Hedlund, M., Shirky, C., O'Reilly, T., Bricklin, D., Anderson, D., Miller, J., Langley, A., Kan, G., Brown, A., Waldman, M., Cranor, L., Rubin, A., Dingledine, R., Freedman, M., Molnar, D., Dornfest, R., Brickley, D., Hong, T., Lethin, R., Udell, J., Asthagiri, N., Tuvell, W., and Wiley, B. (2001). Peer-to-peer harnessing the power of disruptive technologies.

[MIT-LCS, 2004] MIT-LCS (2004). MIT Oxygen Project, pervasive human-centered computing.

[Osborne, 1997] Osborne, M. J. (1997). Introduction to tutorial on the theory of the firm and industry equilibrium.

[Parashar, 2004] Parashar, M. (2004). Peer-to-peer networks + an introduction to jxta.

[PlanetLab Consortium,] PlanetLab Consortium. PlanetLab, an open platform for developing, deploying and accessing planetary-scale services.

[Priyantha et al., 2000] Priyantha, N. B., Chakraborty, A., and Balakrishnan, H. (2000). The Cricket location-support system.

[R. H. Halstead, 1985] R. H. Halstead, J. (1985). MULTILISP: a language for concurrent symbolic computation. In *ACM Trans. Program. Lang. Syst., 7(4)*, pages 501–538.

[Ratnasamy et al., 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. (2001). A scalable content-addressable network.

[Rowson and Druschel, 2001] Rowson, A. and Druschel, P. (2001). Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems.

[Satoh, 2002] Satoh, I. (2002). Physical mobility and logical mobility in ubiquitous computing environments. *Springer-Verlag Berlin Heidelberg*.

[Shiaa and Aagesen, 2002] Shiaa, M. M. and Aagesen, F. A. (2002). Mobility management in a plug and play architecture. In *IFIP WG6.7 Workshop and EUNICE Summer School on Adaptable Networks and Teleservices*.

[SUN Microsystems, 2005] SUN Microsystems (2005). Java 2 micro edition.

[Susan Crosse and Smith, 2003] Susan Crosse, Elaine Wilson, A. W. D. C. and Smith, C. (2003). P2p.

[Tolman, 2003] Tolman, C. (2003). A fault-tolerant home-based naming service for mobile agents. Master's thesis, Rensselaer Polytechnic Institute.

[Valentin Mesaros and Roy, 2004] Valentin Mesaros, B. C. and Roy, P. V. (2004). P2ps: Peer-to-peer development platform for mozart.

[Van Custem and Mostinckx, 2004] Van Custem, T. and Mostinckx, S. (2004). A prototype-based approach to distibuted applications. Master's thesis, Vrije Universiteit Brussel.

[Varela, 2001] Varela, C. (2001). *Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination*. PhD thesis, U. of Illinois at Urbana-Champaign.

[Varela and Agha, 2001] Varela, C. and Agha, G. (2001). Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices 36, 12*, pages 20–34.

[Venners, 1997] Venners, B. (1997). The architecture of Aglets. *Java World*.

[Wang, 2003] Wang, M. (2003). Manet global connectivity and mobility management using hmipv6 and olsr. Master's thesis, Carleton University.

[Ward, 1998] Ward, A. (1998). *Sensor-driven Computing*. PhD thesis, Cambridge University Engineering.

[Weiser, 1991] Weiser, M. (1991). The computer for the twenty-first century. *Scientific American*, pages 94–100.

[White, 1996] White, J. (1996). Telescript technology: Mobile agents. *J. Bradshaw,editor, Software Agents*.

[Wikipedia, 2004] Wikipedia (2004). Wikimedia foundation project.

[Wilson, 2002] Wilson, B. J. (2002). Jxta.

[Zimmerman, 1996] Zimmerman, T. (1996). Personal area networks. *IBM Systems Journal*.