REME-D: a Reflective Epidemic Message-Oriented Debugger for Ambient-Oriented Applications

Elisa Gonzalez Boix
egonzale@vub.ac.be

Carlos Noguera
cnoguera@vub.ac.be

Tom Van Cutsem
tvcutsem@vub.ac.be

Wolfgang De Meuter
wdmeuter@vub.ac.be

Theo D'Hondt
tjdhondt@vub.ac.be

Software Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT

Debuggers are an integral part, albeit often neglected, of the development of distributed applications. Ambient-oriented programming (AmOP) is a distributed paradigm for applications running on mobile ad hoc networks. In AmOP the complexity of programming in a distributed setting is married with the network fragility and open topology of mobile applications. To our knowledge, there is no comprehensive debugging approach that tackles both these issues. In this paper we present REME-D, an online debugger that integrates techniques from distributed debugging (event-based debugging, message breakpoints) and proposes facilities to deal with ad hoc, fragile networks – epidemic debugging, and support for frequent disconnections. A prototype for REME-D is implemented for the AmbientTalk language using the meta-actor protocol provided by AmbientTalk to implement its features.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids; distributed debugging*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages; object-oriented languages*

Keywords

debugging tools, distributed object-orientated applications, event-loop concurrency, reflection, mobile networks

1. INTRODUCTION

Debugging software is an integral part of the development of any application. Debuggers aid in this task by allowing developers to retrace the execution of a program, looking for places in which it deviates from the intended behaviour. This task, which in sequential programs is already difficult, is further complicated in a distributed environment [2].

When debugging a distributed program, developers must deal with the inherent non-determinism of concurrent processes. This complicates the debugging task since an error detected on a run might not manifest itself in the debugging session. Furthermore, the mere presence of the debugger might exacerbate this non-determinism by affecting the way in which the program behaves. Computations performed by the debugger —recording state, checking for breakpoints, etc.— may affect the order in which processes are executed, making the reproduction of a rare erroneous condition even rarer. This condition akin to the Heisenberg Uncertainty principle, is known as the *probe effect* [7, 14].

In this paper, we focus on providing debugging support for *ambient-oriented applications*: distributed applications running on mobile networks that are built on the ambient-oriented programming paradigm [3](AmOP). Ambient-oriented programming extends the object-oriented paradigm with a set of abstractions to deal with the hardware characteristics of mobile ad hoc networks, namely, the fact that network disconnections are frequent, and devices can appear and disappear as the user moves about (i.e., the network is open). A central principle in the AmOP paradigm is that all distributed communication is *non-blocking*, i.e asynchronous. Ambient-oriented applications thus employ a concurrency model without blocking communication primitives (e.g. the actor model [1], event loop concurrency [15], etc).

In order to provide debugging tools for ambient-oriented applications, two challenges need to be addressed:

Message-oriented debugging. In non-blocking concurrency models, the non-determinism is limited to the order in which asynchronous messages are processed as a message is executed atomically. On the other hand, the distance between the cause of an error and its manifestation (i.e., error latency [2]) can be larger. In sequential programming, a call stack trace is often used to establish a *happened before* relation [13] between function calls. In a non-blocking concurrency model, at the beginning and end of processing each message, the call stack is always empty. This means that there is no trace of the path taken to reach the current execution point outside of a process; thus inter-process communication history is lost. It is precisely this inter-process communication that is essential to understand the behaviour of a distributed application. A debugger must thus provide functionality to trace message passing between communicating parties.

Open debugging sessions. The hardware characteristics inherent to mobile ad hoc networks must be taken into account in the debugging process. Given the dynamic nature of mobile ad hoc networks, a debugging session will consist of a undetermined, fluctuating number of devices. Because of this, a debugger must be able to dynamically add and remove devices from the session. Furthermore, the debugger must allow devices to leave the debugging session without affecting the rest of the participants.

Although a diverse spectrum of debugging tools exists for message-passing programs [5, 8, 18, 21], we observe that no debugger exists offering support for debugging applications running in the highly dynamic environment to which ambient-oriented applications are exposed. The distributed debugging techniques and debuggers developed to date have either been designed for parallel computing (e.g. p2d2 [11], replay [8], TotalView [8]), or for general-purpose distributed computing (e.g. Causeway [18], Millipede [19]). None of these debuggers have been explicitly designed for applications running on mobile networks.

In this paper, we introduce REME-D[1] —for Reflective, Epidemic MEssage-oriented Debugger—, a breakpoint-based debugger that adapts notions of sequential debugging, such as step-by-step execution and state introspection, to ambient-oriented debugging. REME-D combines these features from sequential debuggers with a message-oriented architecture based on event-driven debuggers [5, 11, 17, 18, 21], resulting in a simple debugging toolbox. In order to deal with the dynamic nature of the debugging session, in REME-D encountered devices are "infected" with the debugging session, thus terming REME-D an *epidemic* debugger. We implemented a prototype REME-D for the AmbientTalk language [20] (a distributed object-oriented language designed for mobile ad hoc networks) using its reflective capabilities.

The remainder of this paper is structured as follows: Section 2 discusses existing distributed debugging approaches and points out the features that are useful for debugging ambient-oriented applications under the event-loop concurrency model. In section 3, we introduce the domain of ambient-oriented programming and the event loop concurrency model. Section 4 presents REME-D, explaining the rationale behind each of the features it offers to developers. REME-D's implementation is discussed in section 5, while section 6 concludes the paper and discusses future work.

2. RELATED WORK

Several techniques exist for debugging distributed, concurrent, and parallel programs. Although none of the existing approaches provide support for message-oriented debugging and open debugging sessions, they provide the foundation on which we build a debugger for ambient-oriented applications. Along with our discussion, we highlight the techniques that influenced the design of REME-D.

Debuggers can be categorized in two main families: log-based debuggers (also known as *post-mortem* debuggers) and breakpoint-based debuggers (also known as *online* debuggers). Post-mortem debuggers insert log statements in the code of the program to be able to generate a trace log during its execution. Developers can afterwards browse it to examine the behaviour of the program execution after its completion. Breakpoint-based debuggers, on the other hand, execute the program in a *debug* mode that allows programers to pause/resume the program execution at certain points, inspect program state, and perform step-by-step execution. In this work, we take a breakpoint-based approach as its features provide a simple but fundamental debugging toolbox. Stack traces, for example, give the developer an idea of what has happened before in the execution of the program, giving hints of how the developer got to the current point in the execution. Despite the fact that the stack view does not show total causality, in most cases tracing back the stack is enough to find the cause of a bug [18]. When this does not uncover the cause, breakpoints make it easier to mark to interesting places in the execution. While log-based debugging have been previously explored for message-oriented debugging [18], this paper explores breakpoint-based debugging as it provides a simple yet flexible solution which aligns well with the dynamic nature of ambient-oriented applications.

The main focus of distributed debugging is typically inter-process communication where processes are considered to be the basic building blocks [2]. In order to deal with bugs internal to a process, most of the approaches delegate them to dedicated sequential debuggers [2]. Others offer some support for intra-process communication [4, 11, 18, 19]. In this work, we take the former approach, focusing on inter-process communication based on the use of message passing.

A great body of concurrent and parallel debugging techniques are event-based. Event-based debuggers [14] conceive the execution of a program as a sequence of events. The debugger records the history of the events generated by the application, which can then be used to either browse the events once the application is finished [5, 18], or to replay the execution to recreate the conditions under which the bug was observed [4, 17]. Event-based debuggers have been mainly criticized because the recording process is costly and browsing event history does not scale since manually inspecting huge traces becomes cumbersome and difficult [14]. However, they fit well with a non-blocking concurrency model as message sends and receipts can be represented as separate events. A partial order of such events would accurately reflect the behaviour of a distributed application. Some approaches explore a partial order of the event history based on the *happened before* relation for browsing [18] or replay [8,11, 17]. The happened before relation shows how events potentially affect each other [13], allowing developers to identify potential places that caused a bug and as such, offering a similar functionality as stack traces in sequential debuggers. Our approach adapts event histories based on the happened before relation to a breakpoint-based debugger by allowing developers to browse causal links for messages in the current execution context.

Several breakpoint-based debuggers have been designed for parallel programs using message passing communication including p2d2 [11], TotalView [8], and Amoeba [4]. These debuggers offer the traditional commands to, e.g. stop, inspect and step-by-step execution of a running program. Some of them allow to set breakpoints on statements of one process (e.g. TotalView) or a set of processes (e.g. p2d2, prism). An interesting alternative to traditional breakpoints is *message breakpoints* [21]. A message breakpoint stops all receiver processes of the next message sent by a process.

[1]read as remedy

The notion of message-breakpoints maps well to the event-loop concurrency model, since it allows to define sensible stepping semantics at message passing level.

In terms of debugging applications deployed on highly dynamic networks, for the most part, approaches assume a stable network infrastructure, and fragile communication channels are assumed to be handled at the application level, i.e., communication failures are seen as an application-level errors. In a mobile setting, it is desirable that the debugger gracefully deals with network disconnections. TotalView [9] supports open debugging sessions by relying on the underlying MPI middleware to manage and connect to newly or independently started processes. Developers can dynamically attach processes running on nodes that execute a TotalView debugging agent to a debugging session. This gives a degree of freedom in the configuration of a debugging session necessary for ambient-oriented applications. In our approach, devices can be included in the debugging session without first deploying special facilities.

3. AMBIENT-ORIENTED PROGRAMMING

REME-D is designed to support debugging of ambient-oriented applications. Previous work has described the set of programming characteristics that distinguish the AmOP paradigm from a classic distributed object-oriented programming [3]. In this section, we describe two characteristics which directly impact the design of the debugger.

Mobile ad hoc networks are typically not administered. This means that there are no global name servers, and as such objects need to spontaneously discover required services. Hence, in the AmOP paradigm communicating parties do not need to know one another's exact address or location to get to know one another. In addition, they must be able to keep an up-to-date view of which acquaintances are (dis)connected (so that they can take explicit action when an acquaintance disconnects). The combination of these mechanisms is known as *ambient acquaintance management.*

In the AmOP paradigm, all distributed communication is *non-blocking*. This allows communicating parties to deal with the impact of intermittent connectivity of devices on the application as their control flow is not blocked upon sending or receiving. An ambient-oriented concurrency model is thus a concurrency model without blocking communication primitives. In this paper we consider an ambient-oriented concurrency model based on the model of the E language's communicating event loops [15], which is itself an adaptation of the well-known actor model [1]. In this model, actors are represented as containers of objects encapsulating a single thread of execution (*an event loop)* which perpetually takes a message from its message queue and invokes the corresponding method of the object denoted as the receiver of the message. The method is then run to completion, which is called a *turn*. A turn is executed atomically, i.e. an actor cannot be blocked while processing a message.

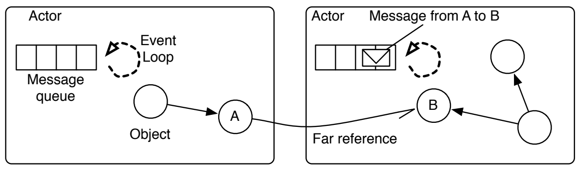


Figure 1: Actors as event communicating loops

Figure 1 illustrates actors as communicating event loops. The event loop (represented by dotted lines) processes incoming messages one by one and synchronously execute the corresponding methods on the actor's objects. Methods can only be directly executed by objects encapsulated by the same actor. Communication with an object in another actor happens asynchronously by means of *far references* (object references that span different actors). For example, when A sends a message to B, the message is enqueued in B's message queue which eventually processes it as shown in Figure 1. A turn thus consists of the execution of a number of synchronous method invocations and asynchronous message sends. Note that at the start and end of a turn, the method invocation stack is always empty. Message return values are then obtained by means of callback objects.

4. REME-D: A REFLECTIVE EPIDEMIC MESSAGE-ORIENTED DEBUGGER

REME-D is a debugger for ambient-oriented programming built on an event loop concurrency model. A prototype of REME-D for the AmbientTalk language has been implemented as the debugger module of the AmbientTalk IDE for Eclipse (IdeAT)[2]. Although the current prototype implementation of REME-D is aimed at AmbientTalk programs, its principles are translatable to other languages built on event-loop concurrency (e.g., E [15]).

To address the challenges posed by a non-blocking concurrency model, REME-D is designed as a breakpoint-based debugger in which focus is placed on the exchange of asynchronous messages between event loops. It adapts features from breakpoint-based debuggers to event loop concurrency –actor state introspection, message breakpoints, stepping over or into turns– , while incorporating for an online usage features from post-mortem, message-oriented debuggers – browsing causal links. To respond to the openness of mobile ad hoc networks, REME-D provides epidemic debugging: it can install itself on newly discovered devices, a process akin to an infection in which REME-D spreads to devices joining the debugging session. Devices can leave the debugging session, either due to communication failures or in response to a user action, without disrupting the debugging of the remaining participants.

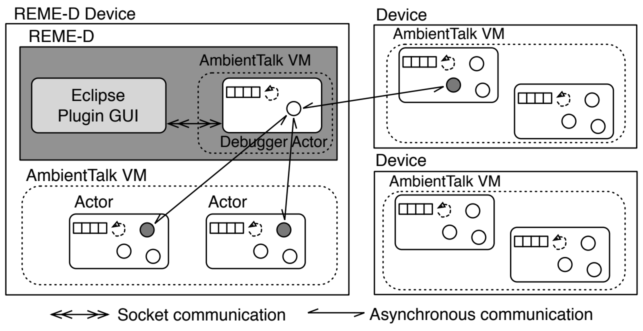
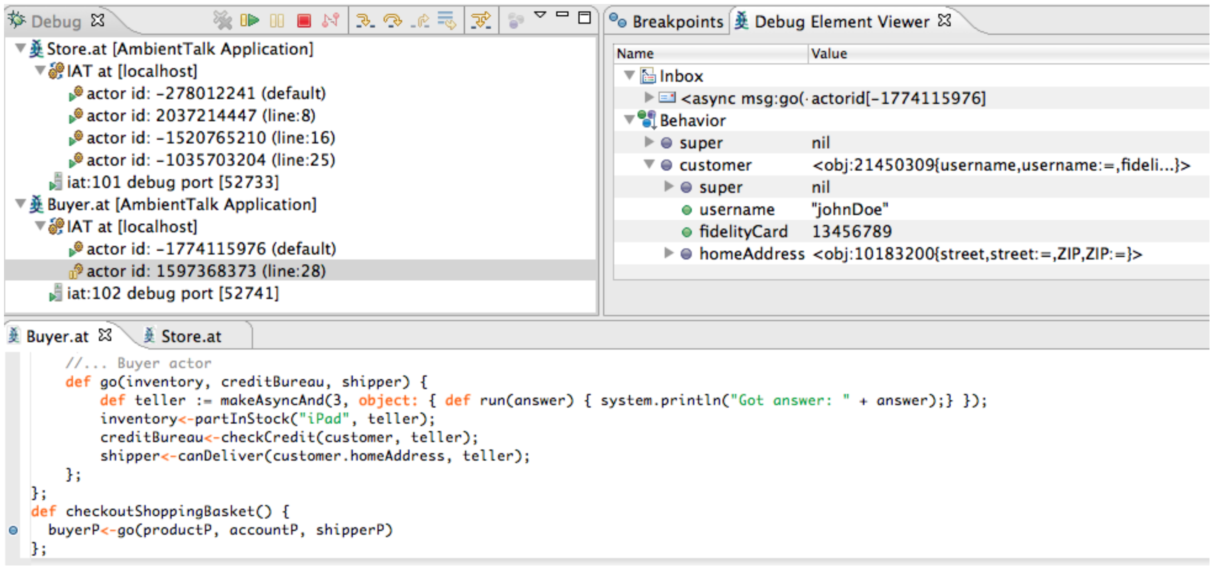


Figure 2: REME-D Architecture Overview

Figure 2 shows the architecture of the REME-D's prototype implementation. It consists of two main components:

[2]The IdeAT plugin is available to be installed from the Eclipse update site at `http://tinyurl.com/ideat`



```
Store.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -278012241 (default)
    actor id: 2037214447 (line:8)
    actor id: -1520765210 (line:16)
    actor id: -1035703204 (line:25)
  iat:101 debug port [52733]
Buyer.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -1774115976 (default)
    actor id: 1597368373 (line:28)
  iat:102 debug port [52741]
```

```
Inbox
  <async msg:go(· actorid[-1774115976]
Behavior
  super           nil
  customer        <obj:21450309{username,username:=,fideli...}>
    super         nil
    username      "johnDoe"
    fidelityCard  13456789
  homeAddress     <obj:10183200{street,street:=,ZIP,ZIP:=}>
```

```
    //... Buyer actor
    def go(inventory, creditBureau, shipper) {
        def teller := makeAsyncAnd(3, object: { def run(answer) { system.println("Got answer: " + answer);} });
        inventory<-partInStock("iPad", teller);
        creditBureau<-checkCredit(customer, teller);
        shipper<-canDeliver(customer.homeAddress, teller);
    };
};
def checkoutShoppingBasket() {
  buyerP<-go(productP, accountP, shipperP)
};
```

Figure 3: Eclipse plugin showing a REME-D debug session.

the coordinator debugger actor (or just *debugger actor*) and the debugger UI implemented on top of the Eclipse Debug Framework [6]. The debugger actor is executed on a different AmbientTalk virtual machine than the application-level actors being debugged. This actor serves as a central manager for all actors within a debugging session. Each actor participating in the debugger session contains a dedicated object (denoted in grey in the figure) to communicate with the debugger actor called *local debugging manager* (or just local manager). Information is transferred between the debugger actor and local managers via asynchronous message passing. Note that communication is bidirectional as actors inform the debugger actor of their current state, and they also must be controlled in response to the user's actions within the debugging session (e.g. set a breakpoint).

REME-D's UI displays the debugging information sent to it by participating distributed AmbientTalk virtual machines, and issues debug commands through the debugger actor. Figure 3 shows a typical REME-D session, in this case the debugging of an online shopping application that processes purchase orders similar to the one found in [18]. The figure displays three views: the debug view on the top right pane, the state inspector on the top left pane, and the editor on the bottom. The state inspector shows the state of a paused actor highlighted in grey in the debug view. Debug view shows that the application is composed of two different devices running the STORE and the BUYER files. The editor shows part of the implementation of the buyer which contains a `checkoutShoppingBasket` method. This method is called by the customer when he decides to purchase some goods. In response to this action, the buyer actor `go` method is called which verifies three things before accepting the order: 1) whether the requested items are still in stock, 2) whether the customer has given valid payment information and 3) whether a shipper can deliver the order in time.

4.1 Viewing actor state

REME-D supports the introspection of actors whenever they are suspended (in a pause state). An actor can only be suspended *between* turns. As turns are executed atomically, allowing actors to be suspended only between turns aligns well with the concurrency model. Because of this, the debugger's probe effect is minimised as turns in the debugged actors remain atomic.

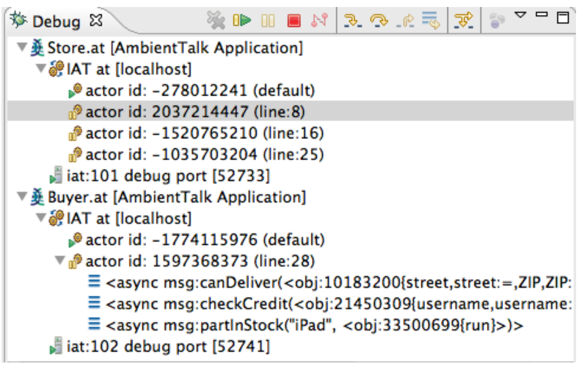
Actors can be explicitly suspended by the developer via a pause command, or implicitly suspended by the local manager as a result of a breakpoint or a step command. When an actor is suspended, the corresponding local manager delays the processing of the message at the head of the queue, until it receives the command to resume execution. The local manager communicates the state of the actor to the debugger actor which in turn updates REME-D's UI. Figure 3 shows, on the top left pane, how this information is presented to the developer in REME-D in the state inspector. The developer is able to inspect the state of the objects encapsulated in the actor, as well as the messages that await processing on the actor's message queue. In this example, the actor contains an object `customer` and a `go` message emitted by an actor with the id `-1774115976`.

While an actor's execution is paused, the state of its objects remains static –no other execution thread has access to them. However, this is not the case for the actor's message queue since other, non-paused actors can still send messages to it. New messages that arrive while the actor is paused are queued, and the local manager notifies the debugger actor to update the state of the actor's message queue. Notice that while an actor is paused, its message queue can only grow, and the order of messages received is not altered.

4.2 Asynchronous Message Breakpoints

In sequential debuggers, breakpoints are placed on statements to mark "interesting points" in the execution of the program in which the developer wishes to inspect its state. Such interesting points in ambient-oriented programming take the form of messages passed between actors leading to the concept of *asynchronous message breakpoints.*

To debug a message, the developer places a breakpoint on the asynchronous message send statement. In AmbientTalk an asynchronous message statement is expressed as `o<-m()` while a synchronous message statement is expressed



```
Store.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -278012241 (default)
    actor id: 2037214447 (line:8)
    actor id: -1520765210 (line:16)
    actor id: -1035703204 (line:25)
  iat:101 debug port [52733]
Buyer.at [AmbientTalk Application]
  IAT at [localhost]
    actor id: -1774115976 (default)
    actor id: 1597368373 (line:28)
      <async msg:canDeliver(<obj:10183200{street,street:=,ZIP,ZIP:
      <async msg:checkCredit(<obj:21450309{username,username:
      <async msg:partInStock("iPad", <obj:33500699{run}>)>
  iat:102 debug port [52741]
```

Figure 4: Debug view after a step-into command

as `o.m()`. In Figure 3 this is signaled in the editor by a dot next to the breakpointed message. When a developer sets a breakpoint, the debugger actor notifies all local managers of the new breakpoint so that when the message is sent, it is annotated with the breakpoint. The actor is suspended when the breakpointed message reaches the head of its message queue, *before* its execution.

REME-D provides the option of selecting at runtime on which devices a given breakpoint will be active. REME-D UI will then contact the debugger actor which, in turn, notifies the corresponding set of local managers of the new breakpoint. This is useful when debugging, for example, peer-to-peer ambient-oriented applications in which a single object plays the roles of both client and server. In these applications, all peers load the same source code so breakpoints will be active on all running peers by default. REME-D also allows the developer to "activate" breakpoints on a per-device basis.

4.3 Navigating Consequence Links

REME-D allows developers to perform a step-by-step execution of a running application. Two kinds of step commands are offered: step-over and step-into a turn.

Stepping over a turn allows the developer to follow the evolution of the actor state as it processes incoming messages. A step-over command instructs the local manager to process a single message –the one at the head of the queue– and return to the paused state.

Stepping into a turn allows the developer to navigate the *consequences* of processing a given message, i.e., the messages sent to other actors in that turn. This is important when understanding the behaviour of an actor since actors cooperate with each other in order to carry out tasks [10]. When the developer instructs REME-D to step into the current turn, the local manager will perform a step-over and mark all outgoing asynchronous messages as breakpointed messages. At the end of step-into, the current actor (i.e., the one on which the command was invoked) and all the actors receiving messages sent within that turn are paused. The developer can then assert the effect of the turn on the current actor, and decide which of the now paused receiving actors to continue debugging. This semantics is reminiscent to the ones provided in [21] by combinaning a message breakpoint with a traditional breakpoint on send statements.

Figure 4 shows the state of REME-D's UI after having stepped into the turn that processed the `go` message shown in Figure 3. The debug view shows the actor that performed the turn is expanded displaying the messages emitted during the turn (`canDeliver`, `checkCredit` and `partInStock`), and the three paused actors that received them.

4.4 Browsing Causal Links

In sequential debuggers, the call stack gives the developer an idea of how the application has reached its current state. Sadly, in the event-loop concurrency model, the call stack is emptied at the end of each turn, providing no information to the debugger. Since all inter-actor communication is performed through asynchronous message passing, a traditional call stack would be of no use in establishing the history of the distributed behaviour of the application.

REME-D takes an event-driven approach by keeping a history of the messages sent and received during a turn. It is possible to query the turn from where a message originated and the message that was being processed in that turn, thus establishing a happened before relation between messages. The developer can then interactively unravel the *causal links* that led to the currently inspected message. Local managers maintain the event history for each actor, noting per each incoming message (processed in a turn) all outgoing messages. This information is then sent to the debugger actor which is in charge to build the causal links for a particular message being browsed.

4.5 Debugging in face of partial failures

As REME-D is geared towards debugging ambient-oriented applications, it is subject to the hardware characteristics of any other ambient-oriented application, namely frequent disconnections and the fact that the network is open. This section describes how REME-D deals with the former characteristic, and the latter is described in the next section.

In order to deal with the frequent disconnections inherent to mobile ad hoc networks, communication with REME-D is non-blocking: REME-D sends debug commands and receives events from participating actors via asynchronous messages. Events are produced either in response to the execution of debug commands e.g., setting a breakpoint, or to inform REME-D of changes in the actor state e.g., new messages enqueued while in a paused state. As such, communication failures with debugged devices do not affect the debugging of the other devices. When a local manager detects a communication failure with the debug actor, it removes the breakpoints, and resumes the actor if necessary. Since it is impossible to predict if a disconnection is temporary or permanent, REME-D takes a conservative approach and resumes the actor computation removing the local manager.

4.6 Epidemic Debugging

Debugging applications written for mobile ad hoc networks require that the debugging process itself be open. In this sense, REME-D's debugging sessions are not constrained to a fixed configuration. The developer does not need to define a-priori which devices will participate in the session. Instead, REME-D operates in an epidemic fashion, spontaneously adding devices to the current debugging session whenever they interact with other actors. A device is added to the debugging session whenever it receives a breakpointed message from an actor. Whenever a breakpointed message arrives to an actor, the AmbientTalk VM deploys a local manager on the newly infected actor. The local manager then announces its presence to the debugger ac-

`createMessage(name,args,tags)`	Creates a message from name, arguments and type tag annotations.
`send(receiver,message)`	Sends a message asynchronously to the receiver.
`schedule(receiver,message)`	Adds a message to the actor's message queue.
`serve()`	Dequeues a message from the actor's message queue and process it.

Table 1: Reflective Operations overridden by the debugger actor mirror

`@Debug`	Annotation used to mark messages as a debugging command from the debugger actor.
`@Pause`	Annotation used to mark messages that require pausing the receiver actor. It is used both by breakpointed messages and messages sent during step-into command.

Table 2: Annotations on Asynchronous Messages

tor, which adds the actor to the debugging session and sends back debugging information (e.g the active breakpoints).

Devices leave a debugging session when they disconnect from the network. As explained before, if a device was suspended when it disconnected, the local manager resumes the actor. On the other side, the debugger actor removes the disconnected actor from the debug view. In figure 4, the debug view on the right pane shows two different devices running `Store.at` and `Buyer.at` respectively. Newly infected devices would appear in this same pane.

5. IMPLEMENTATION

As previously mentioned, we implemented a prototype of REME-D for AmbientTalk programs. The prototype has been built *reflectively* in the AmbientTalk language itself. The debugger actor has been implemented as an actor while the local debugging manager as a *meta-actor protocol*. A meta-actor protocol is similar to a meta-object protocol [12], but it allows developers to introspect on or alter the default semantics for an actor instead of an object. In AmbientTalk, a meta-actor protocol is implemented as a special type of object called an *actor mirror* [16]. In the rest of this section we sketch the implementation of REME-D's features.

Debugger actor. The debugger actor keeps an up-to-date list of connected actors in the debugging session. The list is updated whenever an actor loses connectivity by registering a callback that is invoked whenever a device disconnects from the network. In response to a user's action, the debugger actor sends an asynchronous message to the corresponding local manager(s). Those messages are annotated with a `Debug` annotation so that a local manager can distinguish between application-level messages and debug-level messages. Annotations used in the REME-D prototype are shown in Table 2. When a user sets a message breakpoint in the UI, the debugger actor informs all local managers about the source line corresponding to the send statement.

Debugger actor mirror. The debugger actor mirror is an actor mirror representing the local debugging manager. It implements the necessary interface methods for each debugging command in order for the debugger actor to control the actor. In addition, it alters the default language semantics for message sending and receiving to implement the described REME-D features. Table 1 shows the list of methods that this actor mirror overrides to this end [3].

The `createMessage` and `send` methods reify the default semantics for sending of asynchronous messages of an actor. The debugger actor mirror overrides the `createMessage` method to add a `Pause` annotation (c.f. table 2) in a message to be able to pause the receiver actor. A message is also extended to include information about the sender object in order to build the event history for browsing causal links. The `send` method was overridden to notify the debugger actor about messages being sent from an actor.

The `schedule` and `serve` methods, on the other hand, reify the default semantics for message receiving. The `schedule` method is called right before a message is added in the message queue of an actor. It is overridden to implement the pause command. When an actor receives a pause command, the actor changes its state to paused. If the actor is in a pause state when `schedule` is called, the incoming message is buffered and the debugger actor is notified of the arrival of a new message. The debugger actor in turn updates the UI representation of the message queue in the inspector. This semantics are not applied for debug-level messages. If the incoming message has a `Debug` annotation, the default semantics of schedule are applied and the debugger actor is not notified. The `serve` method is called when a message is dequeued, before being processed. It was overridden to implement the resume and step commands. The first thing that `serve` checks is the message's annotations. If the message has a `Debug` annotation, it is directly processed (as it represents a debug-level message sent by the debugger actor). If the message has a `Pause` annotation, the actor state is changed to pause, and the debugger actor is notified of its suspension. Before processing a message, we check whether a message has a breakpoint. Each message carries the source line number where it was created. In AmbientTalk, this corresponds to the place where the `<-` was used, i.e. the asynchronous message send statement. The method thus checks whether this line number corresponds to any of the ones received from the debugger actor.

Infecting AmbientTalk VMs As explained before an actor becomes infected when it receives a breakpointed message. This has been implemented by altering the default semantics for messages annotated with the `Pause` annotation. In AmbientTalk it is possible, at runtime, to install a new meta-actor protocol on an existing actor overriding an actor's MOP methods. When a message is annotated with `Pause`, the method responsible for processing the message is also overridden to be able to install the debugger actor mirror on the receiver actor. The only requirement for infecting an actor is thus, that the receiving actor knows the source code for the debugger actor mirror. The source code is included in the default AmbientTalk standard library, and thus accessible to any created actor.

Implementation Status Although our REME-D prototype has been built entirely reflectively, it required to change the AmbientTalk interpreter to provide access from the reflective layer to the source line number of asynchronous messages. Currently, causal link browsing is not integrated into the debugger's UI: the event logs produced by REME-D are

[3]For a complete description of the reflective API of AmbientTalk, we refer the reader to a dedicated publication [16].

output into a format readable by Causeway [18].

6. CONCLUSION AND FUTURE WORK

We have presented REME-D, an online, message-oriented debugging approach for ambient-oriented programs under event loop concurrency. REME-D adapts well-known features from sequential debuggers, such as step-by-step execution, state introspection and breakpoints, to the event loop concurrency model. We deal with the communication fragility and the changing network topology inherent to the ambient domain by having the debugger itself be an ambient-oriented meta-program that can reproduce itself on newly discovered devices in an epidemic fashion. Since communication between the debugger manager and the debugged actors happens via asynchronous messages, disconnections of debugged actors do not affect the debugging session. Such a design also minimizes the debugger's probe effect since turns in the debugged actors remain atomic (i.e. REME-D does not block debugged actors) and the order of application-level messages is not altered. To the best of our knowledge REME-D is the first debugger that combines these features together into one debugging toolbox and applies them to ambient-oriented applications. REME-D has been prototyped in the AmbientTalk language and integrated as part of AmbientTalk IDE for Eclipse.

We would like to explore several avenues of future work. First off, since the focus of REME-D is on inter-process communication, debugging intra-actor messages is currently not supported. Including support for such messages would require adaptations to AmbientTalk's interpreter, so that the continuation stack is reified. Secondly, we would like to add a new kind of step command geared towards debugging messages carrying an implicit callback to return computation results (i.e. future-based message passing). This would allow the developer to step through a future-typed message, and have the sending actor be paused again when the returning future is resolved. The reflective implementation of REME-D has the benefit of allowing developers to extend the debugger with new capabilities within the language itself such as support for timeouts, allowing the modification of values on a paused actor, and replaying an execution.

Acknowledgments

Elisa Gonzalez Boix is funded by the prospective research for Brussels program of IWOIB-IRSIB. Tom Van Cutsem is a post-doctoral fellow of FWO-Vlaanderen. This work has been partially funded by the Interuniversity Attraction Poles Programme of the Belgian State through the Belgian Science Policy. We would like to thank Anh Le Tuan who implemented the first debugger for AmbientTalk programs.

7. REFERENCES

[1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] W. H. Cheung, J. P. Black, and E. Manning. A framework for distributed debugging. *IEEE Software*, 7(1):106–115, 1990.

[3] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented Programming in Ambienttalk. In *ECOOP'06*, volume 4067 of *LNCS*, pages 230–254. Springer-Verlag, 2006.

[4] I. J. P. Elshoff. A distributed debugger for amoeba. *SIGPLAN Not.*, 24(1):1–10, 1989.

[5] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation*, pages 271 – 284, Cambridge MA, USA, Apr. 2007.

[6] B. Freeman and B. D. Wright. The eclipse debug framework. EclipseCon tutorial, 2005.

[7] J. Gait. A debugger for concurrent programs. *Software:Practice and Experience*, 15(6):539–554, 1985.

[8] C. Gottbrath. Deterministically troubleshooting network applications. Technical report, TotalView Technologies, Apr. 2009.

[9] C. L. Gottbrath, B. Barrett, B. Gropp, E. R. Lusk, and J. Squyres. An interface to support the identification of dynamic MPI-2 processes for scalable parallel debugging. In *European PVM /MPI Users' Group Meeting*, LNCS, pages 115–122, Germany, 2006.

[10] Y. Honda and A. Yonezawa. Debugging concurrent systems based on object groups. In *ECOOP'88*, LNCS, pages 267–282, UK, 1988. Springer-Verlag.

[11] R. Hood. The p2d2 project: building a portable distributed debugger. In *Symposium on Parallel and distributed tools*, pages 127–136, USA, 1996. ACM.

[12] G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, USA, 1991.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications ACM*, 21(7):558–565, 1978.

[14] C. E. Mcdowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21:593–622, 1989.

[15] M. Miller, E. D. Tribble, and J. Shapiro. Concurrency among strangers: Programming in E as plan coordination. In *Symp. on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, 2005.

[16] S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, and W. De Meuter. Mirror-based reflection in ambienttalk. *Software: Practice and Experience*, 39(7):661–699, 2009.

[17] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, pages 502–511, CA, USA, 1992. IEEE Computer Society Press.

[18] T. Stanley, T. Close, and M. Miller. Causeway: A message-oriented distributed debugger. Technical Report HPL-2009-78, HP Laboratories, 2009.

[19] E. Tribou and J. Pedersen. Millipede: A multilevel debugging environment for distributed systems. In *PDPTA'2005*, volume 1, pages 187–193, 2005.

[20] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, and W. De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Inter. Conf. of the Chilean Computer Science Society*, pages 3–12. IEEE CS, 2007.

[21] R. Wismüller. Debugging message passing programs using invisible message tags. In *European PVM/MPI Users' Group Meeting*, pages 295–302. Springer-Verlag, 1997.