# Distributed Debugging for Mobile Networks

Elisa Gonzalez Boix[a], Carlos Noguera[a], Wolfgang De Meuter[a]

[a] *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 debugging approach that tackles both these issues. In this paper we argue that a novel kind of distributed debugger that we term an *ambient-oriented debugger*, is required. We present REME-D (read as remedy), an online ambient-oriented 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.

*Keywords:* distributed debugging, distributed object-orientated applications, event-loop concurrency, mobile networks

## 1. Introduction

Debugging software is an essential part of the development process of any application. This task, which in sequential programs is already difficult, is further complicated in a distributed environment [1]. 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. The lack of global clock and communication delays makes impossible to determine whether a process is not making progress as expected or has just failed. Furthermore, developing debugging tools for distributed applications is difficult since 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 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* [2, 3].

In this paper, we focus on providing debugging support for *ambient-oriented applications*: distributed applications running on mobile ad hoc networks that are built on the ambient-oriented programming paradigm [4]. Ambient-oriented programming (AmOP) 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. 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 [5], event loop concurrency [6]).

In order to support the construction of ambient-oriented applications, the software development process itself has to become more systematic. Software tools contribute to this task. This has motivated research in integrated development environments (IDEs) and other tools such as debuggers and profilers. Nowadays developers typically edit, compile and debug their programs in a single integrated environment. Distributed applications, in particular, ambient-oriented applications are not different in this regard. However, the omnipresence of failures in mobile ad hoc networks requires us to rethink the design and implementation of software tools. This work therefore investigates tool support for MANET applications in the form of a debugger that handles partial failures. Since partial failures may percolate from the underlying distributed system layers up to the graphical user interface of an application, the need arises for managing partial failures up to the tool level.

Distributed debugging techniques and the debuggers developed to date have either been designed for parallel computing (e.g., p2d2 [7], TotalView [8], Node Prism [9]), for grid computing (e.g., Net-Dbx [10], and IC2D [11]), or for general-purpose distributed computing in fixed, stationary networks (e.g., Amoeba[12], Causeway [13], and Millipede [14]). None of these debuggers have been explicitly designed for applications running on mobile networks. They lack the necessary features to deal with the difficult task of debugging distributed asynchronous applications which run on a radically different network topology, in particular, to deal with the effects of partial failures. After all, debugging requires a thorough understanding of the application being debugged, as well as the programming model on which it is built. Because of this, we claim that a new kind of debugger

is required specifically for ambient-oriented applications.

In this paper, we present an *ambient-oriented debugger*: a distributed debugger that must support the characteristics of AmOP (non-blocking, distributed communication and inherent concurrency) while catering for the constraints of the ambient environment (frequent disconnections, mobile participants), and managing the intrinsic difficulties of writing a debugger such as the probe effect. We then introduce REME-D —for Reflective, Epidemic MEssage-oriented Debugger—, an implementation of this idea in AmbientTalk [4] (a distributed object-oriented language designed for mobile ad hoc networks). REME-D is a breakpoint-based debugger that adapts the 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 [7, 15, 16, 13, 17]; resulting in a simple, familiar but powerful 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.

The rest of this paper is structured as follows. Section 2 illustrates the difficulties ambient-oriented applications by means of a running example and identify the challenges in ambient oriented debugging. Section 3 sketches the requirements for ambient-oriented debuggers and proposes a reference architecture. These requirements are realized in REME-D, our proof-of-concept ambient-oriented debugger for AmbientTalk presented in section 4. Relevant aspects of the implementation of REME-D are presented in section 5. In order to obtain a first assessment of the utility our debugger we conducted a user-study discussed in section 6. After discussing related work, section 8 presents a summary of the paper and discussion on our approach.

## 2. Motivation

Before describing the features of an ambient-oriented debugger, we highlight the need for such a technique by discussing the challenges of debugging MANET applications. To this end, we use an application scenario that we will also use as the running example throughout this paper.

### *2.1. Running Example: the Mobile Shopping Application*

Consider an adaptation of the scenario of the shopping application found in [13] that runs on mobile devices. When the user checks out the shopping cart, the

application implements a protocol for handling purchase orders similarly to well-known shopping websites such as `amazon.com`. Before the shop can acknowledge an order, it must verify three things: 1) whether the requested items are still in stock, 2) whether the customer has provided valid payment information and 3) whether a shipper is available to ship the order in time.

Figure 1 gives a graphical overview of the checkout protocol (verifying the aforementioned requirements) modelled via a distributed object-oriented system where communication between devices is asynchronous. For simplicity, we use explicit callback objects to return the result of an asynchronous computation. When the user check outs the shopping cart in the shopping application UI, the `checkoutCart` message of the service object on the user's smartphone is sent which in turn sends `go` to the user's session object created in the buyer process at the shop. In response to a `go` message, the buyer sends out three messages to the inventory, the credit bureau, and the shipper services called `partInStock`, `checkCredit` and `canDeliver`.

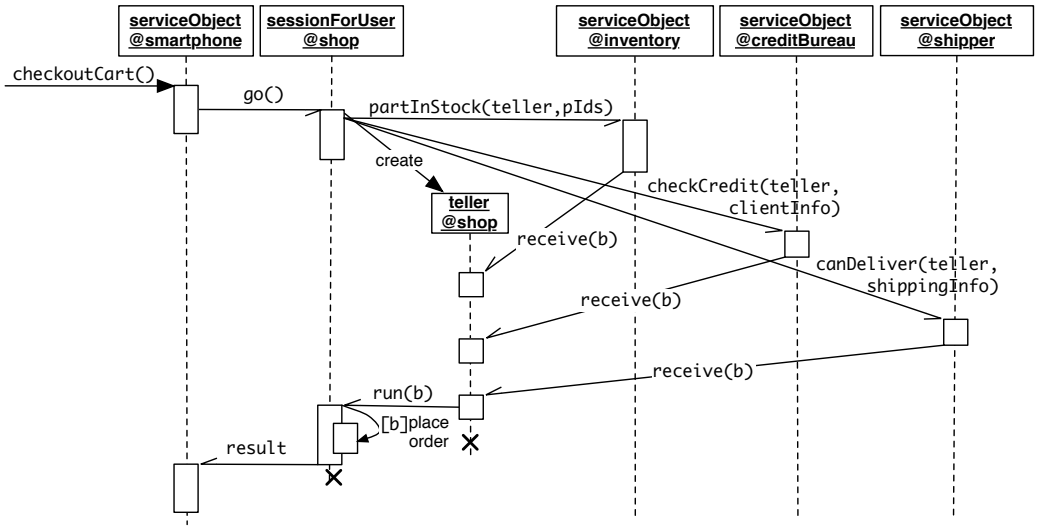



Figure 1: The shopping checkout protocol.

The `teller` object is created and passed as an argument in each of the above mentioned messages, serving as a callback object to collect the answer of the three services. A teller actually is an abstraction implementing an asynchronous adaptation of the logical `and` operator. It is initialized with a number indicating how many affirmative replies it should receive, and the callback object to notify. In this example, the teller is initialized to 3 replies, and the callback object to notify is

the session object residing at the buyer. Once the teller receives the three expected replies, it sends back to the session object a `run(true)` message if all received replies were `true`; otherwise, `run(false)`. The buyer then places the order only if all the requirements become satisfied. Once the order has been placed, the buyer contacts a warranty broker to propose a warranty for the purchases item to the client.

### *2.2. Challenges of Debugging Ambient-Oriented Applications*

Debugging distributed applications is hard because it is difficult to determine the what caused a bug because it may affect or depend on many nodes in the network or specific sequences of messages between the nodes. For example, consider a bug manifests itself in the mobile shopping application when returning an erroneous result for the `checkoutCart` message in Figure 1. In order to find what caused the bug, one can use a distributed debugger to start examining the execution of the `run` message from the shipper (as it denotes the request that produced the erroneous result). In the worst case, one also needs to examine the `receive` messages from the shipper, credit bureau and inventory processes, and so on. Despite being a small example, this may already imply the inspection of 4 different nodes, and the understanding of the whole shopping checkout protocol.

Debugging ambient-oriented applications is even harder because of the radically different nature of the network topology in which they run and the programming model on which are built. In a non-blocking programming model, each received message is processed to completion (there are no blocking receive operations), nodes are subject to data deadlocks (a node will not hang due to a race condition but it may not make any progress because it requires the answer of another one), and relevant parts of the application involve *pipelining* [18] (if a particular component generates returns incorrect results, other components may not detect it immediately when messages are pipelined). The network topology of mobile ad hoc networks, on the other hand, incurs in a high ratio of unanticipated network failures (due to device mobility) which further complicates the debugging process because nodes may disconnect and reconnect while executing/debugging an application, and the faulty nodes may not be present at the time the bug manifests.

These observations has led us to identify two challenges that need to be addressed in order to enable distributed debugging in a mobile environment:

***Message-oriented debugging***. In non-blocking concurrency models, non-determinism is limited to the order in which asynchronous messages are processed since

a message is executed atomically, i.e no external thread can interleave on each instruction while a message is being processed. As such, a debugger should be able to trace asynchronous messages exchange between different processes and allow developers to establish a *happened-before* relation [19] between them. In sequential debugging, a call stack trace is often used to establish a *happened-before* relation between function calls. This aids users to find when an error occurred by answering the question "What likely caused this to happen?" [13]. However, 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. It is also important to note that the distance between the cause of an error and its manifestation (i.e., error latency [1]) can be larger in a non-blocking concurrency model. Recall the bug that manifested itself by returning an erroneous value of the `checkoutCart` message. In this case, the erroneous value is returned in an asynchronous message sent after eight asynchronous messages has been processed as a result of the `checkoutCart` message. In order to find what caused the bug, in the worst case, the entire "happened-before" relation chain must be considered, examining the messages to/from the shipper, credit bureau and inventory processes. In short, a debugger designed for a non-blocking concurrency model must be able to trace message passing between communicating parties, leading to the concept of *message-oriented debugging*.

***Open debugging***. In traditional distributed debugging, the debugger may need to interact with the runtime system in order to manage the different processes which a distributed application consists of [20]. Given the dynamic nature of MANETs, the number of processes that comprises an application is unknown when the debugging session is started. In our running example, one may assume that while debugging the bug manifested on the return value of the `checkoutCart` message, the three processes contacted for placing an order are known. However, once the order has place, the application will search in the network a warranty broker which is discovered in an ad-hoc manner only if the end-user specifies that would like to obtain warranty quotation for certain items. Indeed, ambient-oriented applications typically have to discover and collaborate with other partner applications when they meet in the environment as the device moves about. For example, in a mobile chat application, it is not possible to know beforehand how many participants will participate in a chat. *As a result, a debugging session will consist of an*

*undetermined, fluctuating number of processes according to the applications discovered in the network.* Because of this, a debugger must be able to dynamically add an application to an ongoing debugging session at runtime, i.e., the debugging session must be open. Furthermore, the debugger must also allow objects to leave the debugging session without affecting the rest of the participants as devices may leave the network at any time. Finally, a distributed debugger designed for a mobile environment needs to be able to operate in a deployed mobile application since the correctness of a mobile system may depend on network states as well as the unpredictable mobility of devices.

The above two characteristics have been distilled from the analysis of the implications of the hardware phenomena inherent to mobile networks on the design of a distributed debugger. We henceforth refer to distributed debuggers that adhere to them as *ambient-oriented debuggers* (AODs). In short, an AOD should be able to deal with messages passed between communicating parties and provide control over the flow of asynchronous messages, as well as being able to be dynamically deployed on mobile devices when necessary.

## 3. Overview of an Ambient-Oriented Debugger

We now describe the design and implementation of an AOD built around two central ideas: (1) to adapt features from breakpoint-based debuggers to a non-blocking concurrency model based on the event loop concurrency model [13], and (2) to treat the debugging process as an ambient-oriented application itself which adapts its behaviour to the changes on the network configuration.

We have implemented an AOD for AmbientTalk[4], a distributed object-oriented language designed for mobile ad hoc networks. Although the current prototype implementation of such an AOD is done in AmbientTalk, its principles are independent of the implementation language and could be developed in other languages and software platforms for mobile networks built non-blocking concurrency models (e.g., White [21], iScheme [22]).

*Features of an Ambient-Oriented Debugger*

We detail four major features of our AOD: *state inspection, stepping, causal link browsing, and epidemic debugging*. The later is unique to ambient-oriented debuggers while the others are inspired by features of traditional debuggers adapted to AmbientTalk's event loop concurrency model.

***State Inspection***. An AOD is designed as a breakpoint-based debugger providing users with visibility and control over the target application. The debugger's emphasis is placed on the asynchronous communication. In a non-blocking concurrency model, an application consists of a number of processes (denoted as actors) that execute part of the application, and communicate with other actors by means of asynchronous message passing. These messages are the focus of AOD. When an actor is suspended, users can inspect the actor's state which consists of the objects hosted by the actor as well as the actor's mailbox. An AOD only allows the inspection of actors whenever they are suspended, and this can only happen *between* turns. A turn corresponds to the execution of an asynchronous message and runs till completion before the next message is served. Since turns are executed atomically, allowing actors only to be suspended between turns respects the non-blocking concurrency model.

***Stepping and breakpoints***. To control the debugged application, users can place breakpoints to mark "interesting points" in the execution of the program at which the developer wishes to inspect the state. In an AOD, such interesting points take the form of messages exchanged between actors. As such, breakpoints are place on messages rather than on instructions as traditional breakpoint-based debuggers. In a non-blocking communication model, there are two places in which the debugger may check if a message hits a breakpoint: when the actor serves a message that needs to be sent to another actor (i.e., on the actor's outgoing message queue), and when the actor serves a message that needs to be received by one of its objects (i.e., on the actor's incoming message queue). We denote by *breakpointed message* a message which has hit a breakpoint and will pause the actor's execution when it reaches the head of an actor's message queue.

Applying breakpoints on messages it also allows defining meaningful stepping semantics at the message passing level. In AODs, stepping consists of executing the target application one *turn* at a time. As in a sequential breakpointed debugger, three kinds of step commands are offered: *step-over*, and *step-into* and *step-return* a turn. In the next section, we detail the concrete semantics of different breakpoints and step command supported in our concrete implementation of an AOD.

***Causal Link Browsing***. Causal link reconstruction allows the user to browse the history of messages sent and received in a turn. In sequential debuggers, the call stack gives the developer an idea of how the application has reached its current state. Unfortunately, in a non-blocking concurrency model, the call stack is empty

at the end of each turn, thus providing no information to the debugger. Since all inter-actor communication is performed via asynchronous message passing, a traditional call stack is of no use in establishing the history of the distributed behaviour of the application. Rather, a partial order of messages sent and received would accurately reflect the distributed behaviour of the application. An AOD records the exchange of asynchronous messages during the execution of the debugged application, and then lets users browse the obtained message trace.

Maintaing a partial order of message sent and received provides an order of *activation* of computations similar to the call stack in sequential debugging. However, the root cause of a bug may be not accessible anymore on the call stack when a bug manifest itself [23]. In order to assist the process of finding the root cause of a bug, AODs adopt an event-driven approach and also records the history of turns generated by the application. Users can query the turn from where a message originated and the message that was being processed in that turn, thus establishing a *happened before* relation between messages. The developer can then interactively unravel the *causal links* that led to the currently inspected message.

***Epidemic Debugging***. One of the most distinctive features of AOD is its ability to respond to the dynamic nature of MANETs namely frequent disconnections and the lack of infrastructure. In order to deal with such hardware characteristics specific to the mobile environment in which applications are being debugged, AOD should itself be built as an ambient-oriented application and rely on non-blocking communications to control the actors participating in a debugging session.

In addition, an AOD provides *epidemic debugging* which allows the debugging infrastructure to be dynamically installed on newly discovered actors, a process akin to an infection in which the debugger spreads to devices joining the debugging session. As a result, applications can take part in a distributed debugging session without having to explicitly be configured as a participant beforehand. Devices can leave the debugging session at any point in time —either due to communication failures or in response to a user action— without disrupting the debugging of the remaining participants.

### *3.1. Architecture*

State inspection, stepping, causal link browsing, and epidemic debugging allow our AOD to successfully deal with the challenges of debugging ambient-oriented applications explained in Section 2.2. Firstly, state inspection allow developers to assert the state of the program, by exposing the state of the objects within an event-loop. Stepping and causal link browsing respond to the challenge

of message-oriented debugging. As ambient applications follow a non-blocking concurrency model, the passing of messages becomes the determining factor in the control-flow of the distributed application. As such, stepping allows developers to control the flow of messages from one node in the application to another; while causal link browsing allows the developer to *back trace* a message to its originating node. This control must be performed under the networking realities of ambient-oriented applications, i.e., participating nodes can frequently disconnect and reconnect. This means that the AOD itself must cope with the second challenge: open debugging. To address this, the communication between the debugger and the application nodes must be done in an ambient-oriented manner. Furthermore, as *new* nodes might be spontaneously discovered by the application, the AOD must adapt to include them in the debugging session; that is to allow developers to inspect the state and received messages, and control the execution of nodes that were not present in the network when the debugging session commenced. Epidemic debugging is the feature that enables our AOD to cope with this challenge.

Figure 2 gives an overview of the architecture of the ambient-oriented debugger. When debugging an ambient-oriented application, there may exist several devices running parts of the application. As such, debugging support will be distributed over two or more devices. In this case, Figure 2 shows three devices, two of which have joined the debugging session. The device which starts the debugging session of an application is called the *debugger device*. A device which joins the debugging session at a later point in time is called an *infected device*. Finally, there may be devices in the network which are conceptually running part of the target application, but which do not form part of the debugging session. This can happen because e.g., they are out of communication range of the debugger device, or because they do not run code relevant to the part of the application being debugged. A device can opt out of being debugged, i.e., it will never be infected. Allowing devices to explicitly opt-out of a debugging session prevents the most obvious security issues.

As also shown in Figure 2, the debugger device runs two virtual machines (VM): one hosting the debugging infrastructure, and one hosting the target application. The debugger VM consists of two components: the coordinator debugger actor (or just *debugger actor*), and the debugger front end through which the user can interactively control the target application. Each actor participating in the debugging session contains a dedicated object (denoted in grey in the figure) called *local (debugger) manager* implementing the main debugging features previously described. The debugger actor serves as a central manager between the debugger

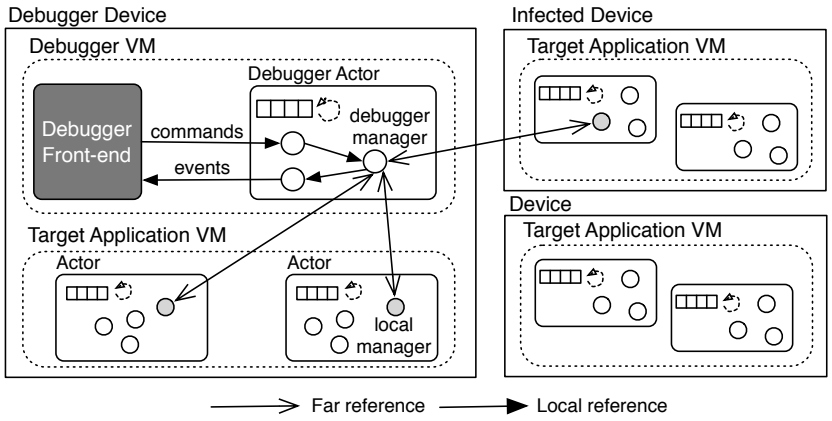



Figure 2: An ambient-oriented debugging session distributed over two devices.

front end and all actors participating in a debugging session. Communication between the debugger manager and the local managers is bidirectional and happens via asynchronous message passing. As usual, users can control a debugged application via the debugger front end, which issues debug *commands* in response to the user's actions (e.g., set a breakpoint, etc.). In response to those commands, the corresponding local manager perform some action (e.g., pausing the actor execution) and inform the debugger actor of their state by sending debug *events*.

## 4. REME-D: an Ambient-Oriented Debugger in AmbientTalk

We now describe a concrete prototype implementation of an ambient-oriented debugger in AmbientTalk called *REME-D*. REME-D is a Reflective Epidemic MEssage-oriented Debugger that has been implemented as the debugger module of the AmbientTalk IDE for Eclipse (IdeAT)[1]. As such, it relies on the Java GUI components provided by the Eclipse Debug API as a front end, and AmbientTalk VMs for the debugger logic. Thus REME-D is itself an ambient-oriented application written in AmbientTalk. REME-D's front end offers three views: actor view, state inspector and editor, as shown in Figure 3.

[1]The IdeAT plugin is available to be installed from the Eclipse update site at `http://tinyurl.com/ideat`, and its documentation is available at `http://tinyurl.com/ideatdocs`

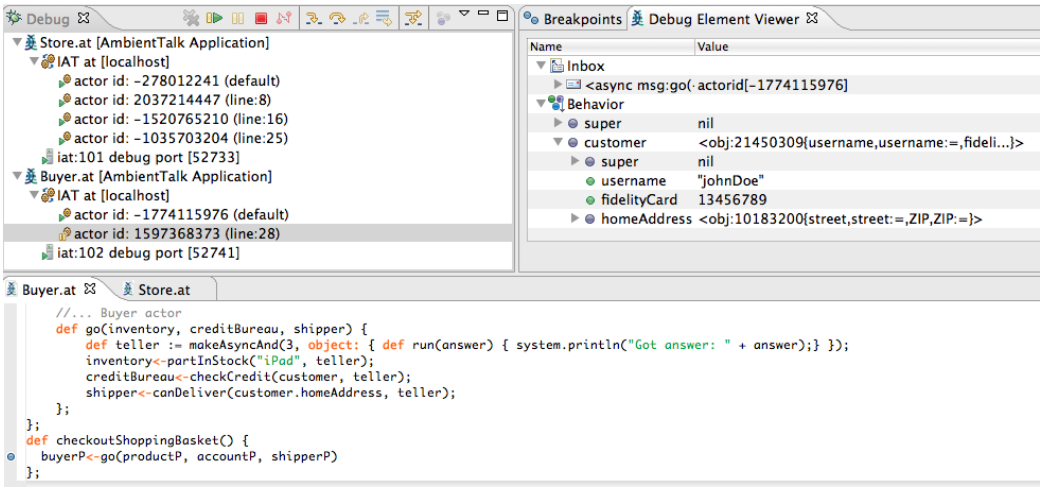



Figure 3: Eclipse plugin showing a REME-D debug session.

### *4.1. Viewing the Actor State*

REME-D supports state inspection of actors whenever they are suspended (i.e., in a pause state). In particular, users can view the state of an actor reachable from the actor's *behaviour* object, and the messages that wait in the actor's message queue to be processed. Notice that while an actor's execution is paused, the state of its objects remains static. This is due to the fact that user interaction with a debugged actor happens *in between turns*. Non-paused actors can still send messages to a paused actor. These messages are queued, and displayed in the paused actor's message queue. The message queue of a pause actor can only grow until it is resumed again. Actor's are paused either by means of a "pause" command (similar to pausing a thread of execution in a traditional debugger), or as a result of a breakpoint activation or a stepping command. When an actor is suspended, the corresponding local debug manager delays the processing of the message at the head of the queue, until it receives the command to resume execution.

Figure 3 (top right) shows the actor state view. In it, the actor contains a `customer` and a `shoppingCart` object, and a `go` message emitted by an actor with the id `-1774115976` awaits processing in the actor's message queue. The developer can interactively unfold each object represented in the actor state view.

### *4.2. Breakpoints Catalog*

REME-D provides a catalog of breakpoints which combines breakpoints on messages with breakpoints existing in sequential debugging. Breakpoints in REME-

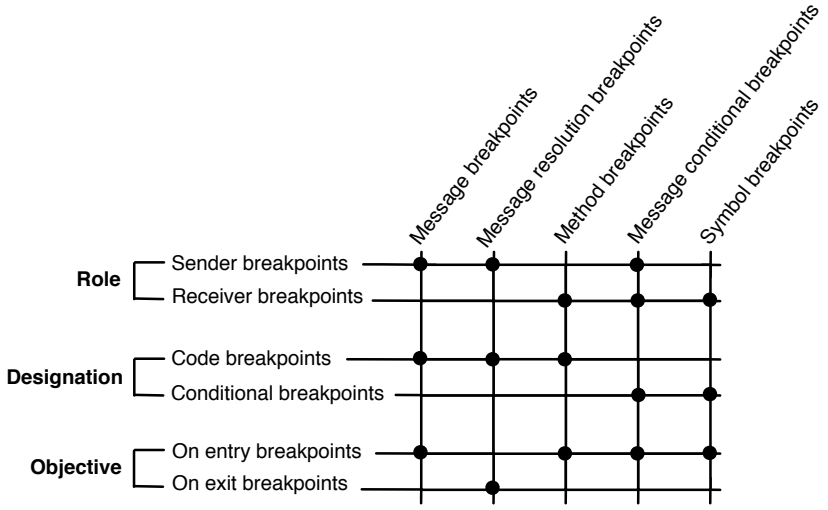



Figure 4: REME-D's breakpoint catalog provided to users.

D can be classified according to three basic properties: role, designation and objective. *Role* determines the place at which the breakpoint is set. We distinguish between breakpoints placed in the actor's outgoing message queue (called *sender breakpoints*), and actor's incoming message queue (called *receiver breakpoints*). *Designation* denotes the way that a user defines a breakpoint. Either in a line of code (called *code breakpoints*), or as a predicate condition about the state of a message (called *conditional breakpoints*). *Objective* denotes when the execution should be suspended. Suspending either the actor's execution when a message reaches the head of the message queue, (a) before the message is executed (called *on entry breakpoints*) and (b) after the message is executed (called *on exit breakpoints*).

Figure 4 places the 5 different breakpoints offered in REME-D in this taxonomy. In this section, we highlight the most relevant breakpoints with respect to ambient-oriented debugging. The whole breakpoint taxonomy can be found elsewhere [24].

**Message breakpoints** A message breakpoint defines a breakpoint on a line of code of an asynchronous message send. In Figure 3 this is indicated in the editor by a blue dot next to the `go` asynchronous message. The actor's execution pauses when the breakpointed message reaches the head of the message queue, *before* the receiver invokes the method corresponding to the asynchronous message.

**Message resolution breakpoints** A message resolution breakpoint defines a

breakpoint on a line of code of an future-type message send, `o<-m()@FutureMessage` in AmbientTalk. The actor execution pauses when the message carrying the return value of the computation reaches the head of the message queue of the sender actor. This means that execution is paused after the future-type message is processed but, *before* the sending actor processes the message with the return value of the computation.

**Message conditional breakpoints** A message conditional breakpoint defines a breakpoint on a conditional expression about a message. It allows users to stop execution whenever the result of an expression is true, without having to predict a particular message send or reception where this may happen. This expression is a boolean predicate over a message in the queue of an actor and the receiver of the message. The actor's execution pauses when the result of the conditional expression is true for a message that reaches the head of the given message queue(s), before the message is processed.

Peer-to-peer is a recurring architecture in ambient-oriented applications in which a single object plays the roles of both client and server. Since all devices have the same source code, REME-D provides support to specify on which device a breakpoint should be active. The developer can set particular devices for which each breakpoint is active manipulating the breakpoint properties through the UI.

### *4.3. Stepping*

As in a sequential breakpointed debugger, REME-D offers three kinds of step commands: step-over, and step-into and step-return a turn. In addition, we provide a variation of step-over called step-until.

**Step-Over** Stepping over a turn allows the user to observe how the state of the actor changes as it processes incoming messages. A step-over command instructs the local manager to process a single message —the one at the head of the queue— and return the actor to the paused state. In addition, the local manager keeps track of all outgoing messages that should be sent during that turn, allowing the user to inspect them.

**Step-Into** Stepping into a turn allows the user to navigate the *consequences* of processing a given message, i.e., the messages sent to other actors in that turn. When the user 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. As a result, at the end of step-into, the current actor (i.e., the one on which the command was invoked) and all the actors receiving messages sent on that turn are paused. Figure 5 shows the debug view after having stepped into a turn, notice all actors that received a message are also paused.

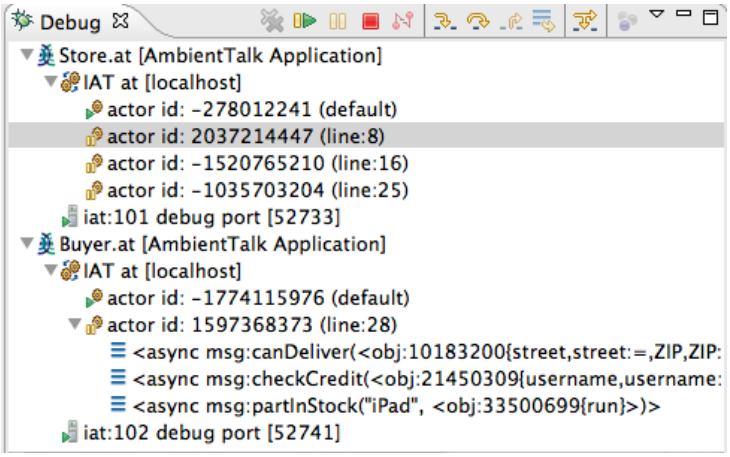



Figure 5: Debug view after a step-into command.

**Step-Return** Step-returning a turn allows the user to return from a message which has been stepped into. When the user instructs REME-D to step return from a future-type message, the local manager will perform a step-over of the message, and mark the message sent with the return value of the computation as a breakpointed message. As a result, at the end of step-return of a future-type message, the actor that *sent* the message is paused when the future associated with the message becomes resolved.

**Step-Until** A step until command takes a conditional expression about a message, and instructs the local manager to pause the execution again when a message that satisfies the given condition is at the head of the message queue. A step-until command is specially handy when debugging distributed interactions in which the same message is sent several times to an object with different state (e.g., taking different argument values).

### *4.4. Browsing Causal Links*

In REME-D, each message contains debugging information about the trace of messages from which the message originated. A trace of messages consists of a list of $< identifier, selector >$ tuples which contains the identifier of the turn in which the message send got created, and the selector of the message being processed in that turn. When a message is sent, the local manager attaches the list of tuples for the message being processed to the message, and then adds a new tuple with the debugging information for that turn.

REME-D allows users to query the turn from which a message originated.

When an actor processes a message from the message queue, the local manager stores a $< id, cause, effects >$ tuple corresponding to the turn identifier, the selector of the message being processed (denoting the *cause* of the turn), and the list of all outgoing messages sent during the turn (denoting the *effects* of the turn).

### *4.5. Open Debugging*

REME-D is an ambient-oriented application in which communication is non-blocking. This means that REME-D sends debug commands and receives events from participating actors via asynchronous message passing. When a local manager detects a communication failure with the debugger manager, it removes all breakpoints and resumes the actor if necessary. This allows REME-D to handle frequent disconnections in a graceful manner.

To debug the behavior of an application when it is disconnected, REME-D provides the user with the possibility of simulating the disconnection of a device. This is achieved by having the local manager on the "disconnected" device cut communication with other devices, while maintaining it with the debugger manager (so that the user can still control the actor from REME-D'S UI).

#### *4.5.1. Epidemic Debugging*

As previously mentioned, in AmOP devices spontaneously discover each other, forming ad hoc networks in which they collaborate to perform a task. Debugging such applications requires flexible debugging sessions, in which the participants of a debug session can change as the distributed application evolves.

To support this, REME-D's debugging sessions are not constrained to a fixed configuration. The user does not need to define a-priori which devices will participate in the session. Instead, REME-D operates in an epidemic fashion, spontaneously adding new devices to the current debugging session whenever they interact with actors participating in the session.

When a debugged device interacts with a device outside the debugging session, REME-D will automatically extend the session so that the new device can be controlled by the debugger. Concretely, upon receiving a breakpointed message, REME-D deploys a local manager to the receiver actor and the VM is said to be *infected*. The local manager then announces its presence to the debugger manager, which adds the actor information to the debugging session and sends back debugging information (e.g., the active breakpoints). Developers can control whether a device is susceptible to infection by means of the `-Xdebug` option.

| | |
|---|---|
| `createMessage(name, args,tags)` | Creates a message from name, arguments and type tag annotations. |
| `send(receiver,msg)` | Sends a message asynchronously to the receiver. |
| `schedule(receiver,msg)` | 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

## 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 AmbientTalk actor while the local debugging manager as a *meta-actor protocol*. An meta-actor protocol is similar to a meta-object protocol (MOP) [25], 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* [26]. Due to space constraints, we sketch the implementation of the debugger actor and the local manager. A comprehensive description can be found in [24].

**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 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 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 actor mirror implementing the local debugging manager, called *debugger actor mirror*, 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 [2].

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.

[2] For a complete description of the reflective API of AmbientTalk, we refer the reader to a dedicated publication [26].

**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, 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.

### *5.1. Discussion*

A side-effect of REME-D's reflective implementation is that it is amenable to scripting using AmbientTalk. Rather than interact with the debugger, scriptable debuggers[27, 28] allow programmers to manipulate the execution of the target program through a script. Scripts consume events emitted by the base program (e.g., function invocation, program termination), and send commands to the manipulate the control-flow of the program or query its state.

REME-D exposes the full debugger API to the programmer. Developers can then leverage this API to either extend the debugger's functionality (e.g., new kinds of breakpoints) or at debug-time execute scripts that automate debugging commands (e.g., stepping an actor until a particular message is processed). The Eclipse plugin has been extended with a console connected to the VM running the debugger manager, called *AT Debugger Manager*. Scripts that automate debugging commands can be launched from within the console connected to the AT Debugger Manager.

The fact that both REME-D and the scripts are implemented in AmbientTalk, also allows the programmer to profit from the event-loop concurrency model offered by AmbientTalk. As previously explained, scriptable debuggers consume events generated by the program under study. As such, many scriptable debuggers [29, 30] resort to call-backs to control the debugger. Debug scripts in REME-D can integrate future-typed messages to send commands to the subject programs, thus reducing the reentrant style of programming that is endemic to callbacks.

The scriptable facilities offered by REME-D open the door to the automatization of debugging tasks normally assigned to developers. One interesting possibility is that of automatic generation of breakpoints [31]. Automatic generation of breakpoints tries to predict interesting points in the execution of the program which might shed a light in the nature of a bug. Zhang et. al. recently propose

an approach called BPGen, which leverages various static analyses to identify possibly buggy statements and tag them with a breakpoint. While REME-D's infrastructure lends itself to such implementations, their realization remains the subject of future work. In terms of automatization, REME-D's implementation is currently limited to the process of infection. As explained in Section 4.5.1, when an actor in the debugging session interacts with (i.e., sends a message to) an actor outside the session, the REME-D runtime automatically includes it in the session, rendering the newly-found actor under the control of the debugger manager.

**Implementation Status.** As previously mentioned, REME-D has been integrated with the AmbientTalk IDE for Eclipse as the debugger module. Nevertheless, REME-D can run independently of this particular front end, e.g there exists another front end written in Java Swing [32][3]. There are a number of features which are not currently integrated in the Eclipse IDE including causal link browsing and simulating network disconnections. These features need to be accessed via the console connected to the AT Debugger Manager.

## 6. Validation

In order to assess the usability of an ambient-oriented debugger, we performed an empirical study. When selecting a methodology to follow for the experimental validation we take into account the following points: First, we are testing a new tool. Second, the AmbientTalk user base is relatively small (estimated to 8 active researchers, and around 100 occasional programmers including master students and outside contributors). Third, we see debuggers as *program comprehension tools*, and therefore for the first empirical evaluation, we conducted a qualitative rather than a quantitate evaluation.

From these observations, we selected a *quasi-experiment* [33], since our experiment complies with the guidelines outlined in [34]. Quasi-experiments allow for the investigation of cause-effect relations, in cases in which randomization is not used. While this sort of experiment does not allow us to make any founded claim regarding the usability of REME-D, it allows us to assess whether participants of the experiment change their view on the difficulty of debugging ambient-oriented applications because of the use of an ambient-oriented debugger. Quasi-experiments have been successfully used in the domain of software engineering as pointed out by Kampenes et. al. [35]. Nevertheless, it is well-known that they

[3] Screenshots of the Swing front end are available at `http://tinyurl.com/al6cfxb`

are subject to concerns regarding the validity of observations from the experiment which we discuss later in Section 6.2 following Kampenes et al. guidelines.

We opt for a *one-group pretest-posttest quasi-experiment design* [33] in our user study. The experiment consists of one group of 22 participants which is subject to a test (*pretest*), that is followed by a series of tasks carried out by the participants. Then the experiment is concluded with second test (*posttest*) [4]. Pretest and posttest employ the same questions varying the independent variable, i.e., introducing REME-D. By comparing the pretest and posttest results, we can measure how the exposure to REME-D influenced the perception of ambient-oriented debugging, and which features of REME-D were deemed useful by participants.

The questionnaires used for the pretest and posttest employ close-end matrix questions in which participants need to rate a number of statements on a five-point Likert scale, (i.e., a 1-5 scale ranging from "totally disagree" to "totally agree"). In order to avoid bias, we intermingle consecutive positive and negative statements.

*Pretest.* The pretest measured the expectations prior to using REME-D. It consists of 22 statements structured along four themes: (1) Participant's background, (2) Development experience(3) Attitude towards debugging and (4) Expectations from an ambient-oriented debugger.

*Debugging Assignment.* Each participant was asked to complete a number of tasks relative to the debugging process of an AmbientTalk application. We employed the mobile shopping application described in Section 2.1. For the purpose of the experiment we seeded two errors, one in the shopping check-out protocol depicted in Figure 1, and one in the interaction with the warranty broker. Both errors are a case of misleading return values errors, as described in [18].

In the first task, the participants were asked to use REME-D to find out why the shopping checkout protocol did not work properly and fix the problem. The second task described the extension to the shopping checkout protocol in which the buyer contacts a (faulty) warranty broker to propose a warranty for the purchases item to the client. Participants were asked to dynamically launch a warranty broken outside the debugging session and interact with it to determine what the problem was.

[4] All the raw data including all the material and 22 filled pre/posttests questionnaires is available at `http://tinyurl.com/av7b5ya`.

*Posttest.* The posttest consists of 24 statements structured along four themes: (1) Assignment experience (2) Value of an ambient-oriented debugger (3) UI Experience, and finally (4) Value of REME-D features, divided in two kinds of questions –how often they used each of the features provided by the debugger, and whether they found them useful. In both pretest and posttest, space was provided for participants to write down comments they had about the tool and the assignment.

### *6.1. Results*

We now discuss the main results from the experiment by first discussing the participant's profile, and then comparing the results of the pretest and the posttest.

#### *6.1.1. Participants profile*

The participants were recruited from within the computer science department of our university, in particular, they were all enrolled in a master (13 participants) or PhD program (9 participants). All participants had previous experience with AmbientTalk. Figure 6 shows a boxplot with the profile of the participants. In general, the participants consider themselves experienced developers (A), although not distributed development experts (B). And, although they are familiar with Eclipse (D), they do not have large experience in the use of its debugger (K).

Figure 7 provides a summary of the participants' attitude towards debugging in a radar diagram[5]. The participants strongly agree that debuggers are a helpful tool to find bugs in programs (I), and in general agree that debuggers aid in the understanding of programs (J). Participants also acknowledge that debugging distributed programs is hard (H), and that better tools can prevent bugs (G). More importantly, participants agree on the need for a debugger for AmbientTalk (P).

#### *6.1.2. Pretest-Posttest*

Overall, REME-D was well-received by the participants. As can be seen in Figure 9a, although most participants were positive with respect to the value of REME-D as a tool to help them find bugs in their programs, their answers in the posttest are more spread. This indicates that although good, REME-D did not meet their expectations. Further discussion with the participants revealed their doubts about the suitability of the assignment, as some of them did not seem convinced that the types of bugs included are representative of real bugs. This is further discussed in the section about threats to validity.

[5]Each branch represents a single question; the bold line shows the average and the colored surface indicates the range of given answers

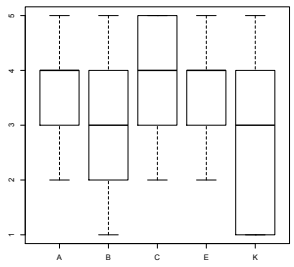



Figure 6: Experience of the participants: (A) development experience (B) distributed development experience (C) understanding of AmOP in AmbientTalk (E) Eclipse experience (K) online debuggers experience.

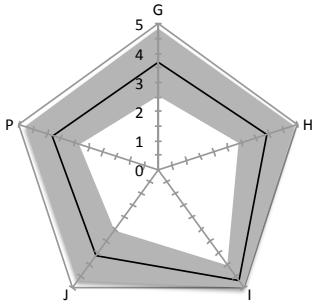



Figure 7: Participants' attitude towards debugging: (G) development tools can prevent a lot of bugs (H) debugging distributed programs is hard (I) debuggers are a helpful tool to find errors in programs (J) debuggers are a helpful tool to understand programs (P) a debugger for AmbientTalk is needed.

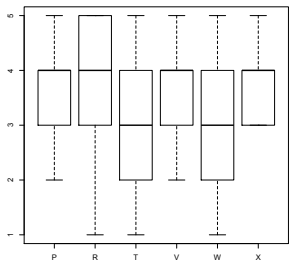
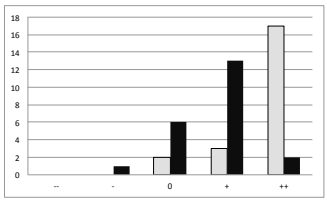
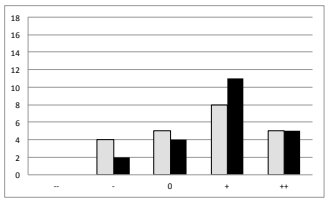


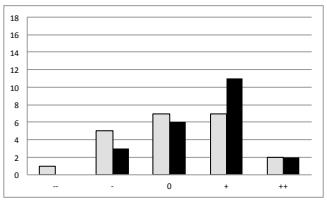

Figure 8: Participants' appreciation of the features of REME-D. (P) message breakpoints (R) step-into command (T) step-over command (V) pause actor command (W) control over program execution (X) infection of VMs.




(a) Finding bugs




(b) Understanding




(c) General easing of programming in AmbientTalk

Figure 9: Comparison of the participants' pretest (grey) and posttest (black) impressions; X axis depicts the 5-point Likert scale, and Y axis is the number of participants that selected each point.

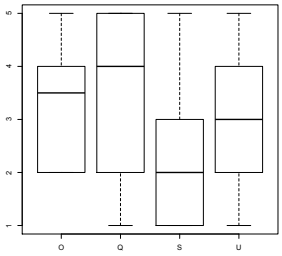



Figure 10: Participants' usage of the features of REME-D. (O) message breakpoints (Q) step-into (S) step-over (U) pause actor.

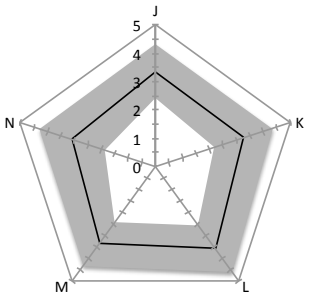



Figure 11: Participants' experience with REME-D's UI in Eclipse: (K) Features are clear and accessible in the UI (L) Actor view gives a good overview of the state of the application (M)Debug Element View gives a good overview of the state of an actor (N) REME-D is helpful but needs a better user interface.

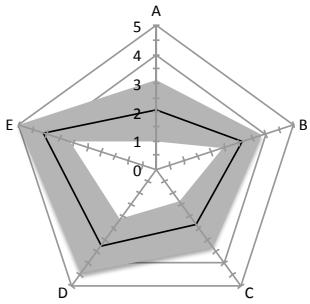



Figure 12: Participants' experience with the assignment: (A) the assignment was too easy (B) the assignment was very interesting (C) the assignment represents the kind of bugs I have encountered in AmbientTalk (D) I would have liked more time to complete the assignment (E) I had enough help in completing the assignment.

Regarding the usefulness of REME-D as a program understanding tool, all participants indicate that the tool helps them to understand AmbientTalk programs (Figure 9b). This is evidenced by seeing a larger number of participants giving higher evaluations in the posttest (black bars are to the left of gray ones).

Similarly, for the use of REME-D as a means to make ambient-oriented programming in AmbientTalk easier (Figure 9c), results show that most participants are more positive in the posttest. It is interesting to remark that 4 out of 22 participants change their opinion on this point after working with REME-D. We regard this result as encouraging since all participants stated in the pretest that they understand the principles of ambient-oriented programming in AmbientTalk.

### *6.1.3. Features of REME-D*

In general (see Figure 8), participants seem to value message breakpoints, step-into command, pause command and the infection of other VMs (all means scored 4). Participants were rather neutral towards step-over command and the

control over the execution of an AmbientTalk program. This seems to be correlated with the low usage of these features, as can be seen in Figure 10.

Regarding the effectiveness of the representation of REME-D's features in the Eclipse IDE, the opinions of participants in the posttest were reasonably positive. Figure 11 provides a summary of the participants' experience with REME-D's UI in a radar diagram. Most participants did appreciate the actor and debug element views provided in the Eclipse IDE (L and M). In addition, all participants find REME-D's UI easy to use with the exception of one participant who "strongly disagreed". That participant was totally disappointed with REME-D's UI; he is also the only participant who did not find debugging features clear and accessible in the UI and strongly disagree when asked about the usefulness of the actor and debug element views. Not surprisingly, he is one of the 4 participants that strongly believe that in essence REME-D is helpful, but it requires a better UI.

### *6.2. Threats to validity*

As previously mentioned, a quasi-experiment study does not allow us to make any generalized claims regarding the usability of REME-D. Instead, quasi-experiments do allow us to observe how potential users perceived our tool. However, quasi-experiments are subject to concerns regarding the validity of the observations resulting from the experiment. According to the guidelines for quasi-experimental research designs, we now sketch the threats to internal and external validity of the experiment, as well as the measures taken to mitigate them. Internal validity considers the validity of cause-effect inferences made during the experiment, while external validity considers the validity of generalized inferences (or how wrong we are when making generalized observations from the study).

*Internal Validity.* The analysis of the experiment's outcomes assumes that REME-D is the only factor influencing the dependent variables. However, several factors may have potentially interfered in the participants' perception of REME-D.

First, the introductory demonstration might have biased participants towards using the REME-D's features shown to them. To counter this effect, we explained all features of REME-D but only showed participants where they could find REME-D features in Eclipse, and the basic REME-D views. Participants were told they could use any feature they liked.

Second, the assignment executed by our participants might be too simplistic or hard. To assess this risk, the posttest included a set of questions to measure the participants' experience with the tasks performed in the assignment. Figure 12 provides a summary of the participants' experience with the assignment in a radar

diagram. The results show that participants generally did not find the assignment too hard (question A) with the exception of two participants, and find the experiment interesting to do (question B).

Finally, the duration of the experiment may also have influenced the internal validity. The results, shown in Figure 12, reveal that participants do not seem to have experienced time pressure (D) and were satisfied with the help received to complete the assignment (E).

*External Validity.* A risk exists concerning the composition of the group of participants: since all participants were computer science students or researchers, they might not form a representative sample of software developers. This is mitigated by the different degrees of expertise that the participants reported. Furthermore, participants were allowed to ask questions about the technology involved in the study at any time.

With respect to the task, although it has been used in previous research efforts in debugging [13], the risk remains of it not being representative of a real-world ambient-oriented application. Finally, the bugs seeded in the application may not have been representative of real-world bugs. Considering that we wanted to limit the amount of time necessary to execute the assignment, the risk exists that the assignment did not capture the complexity associated with real-life bugs in ambient-oriented applications. Indeed, results show that participants did not find the assignment to be representative of the kind of bugs they encountered while developing in AmbientTalk (question C in Figure 12).

### *6.3. Discussion*

#### *6.3.1. Experiment Design*

When considering the user evaluation of the tool, we opted for a qualitative rather than a quantitative user study. The reason for this choice lies in our view of debuggers as a program comprehension tool rather than a bug finding one. Indeed, debuggers do not find bugs as bug finding is a testing activity. Rather debuggers allow developer to test hypotheses about the cause of a bug. Because of this observation, the perception of the debugger by programmers is essential to its success. Nevertheless, we acknowledge the utility a quantitative evaluation of REME-D, for example verifying that the cause of a bug is indeed found faster when using the debugger that when not. The evaluation we have performed of the tool allows us to assert REME-D's *suitability* (whether the features of REME-D attack issues that the programmers find problematic), before answering the question of REME-D's *performance* (whether the use of REME-D makes bug-fixing faster). In this

regard, the *one-group pre-test/post-test* experimental design we chose allows us to test out whether participants see the problems REME-D tackles as real by asking questions in the pretest such as "Inspecting the mailbox contents is essential to debugging AmbientTalk programs". At the same time the experiment allows us to test whether the participants believe that REME-D addresses those problems in a satisfactory manner, by asking questions in the post-test such as "The Debug Element View (which displays the mailbox) gives a good overview of the state of an actor".

*6.3.2. Participant's impression of REME-D*

While working with REME-D, participants encountered a number of issues related to REME-D's UI. While we provided work-arounds for these issues, we believe they influenced the participant's perception of the tool. Participants were also slightly confused by the way the tool presents the actor participating in the debugging session in the actor view. Three participants explicitly included in the posttest comments to improve this view, e.g., "actors should get names instead of line numbers".

Interestingly, more than half of participants (12 out of 22) left feedback on the pretest or posttest questionnaires. All the pretest comments are about desired features in the debugger. In contrast, the posttest comments mainly included suggestions to improve REME-D's UI. In particular, 3 participants complained about the stepping functionality (e.g., "stepping-in was not very perfect. I got lost on where the current execution step was [..]"), and three more about the limitations of message state inspection. Other comments refer to extensions to the existing functionalities of the debugger. Most notably, the ability to inspect any kind of object and not only those reachable from the actor's behavior.

Table 3 summarizes the comments left by participants regarding features they would like an AmbientTalk distributed debugger to exhibit. In the table, for each feature, a count of the number of comments that refer to it is made. Also, features on the table are grouped by whether the feature was present in the tool at the time of the study, whether it was implemented in subsequent versions or whether it is still to do.

## 7. Related Work

A lot of research has been conducted in developing debugging tools and techniques for concurrent and distributed systems, resulting in a large number of tools.

| Feature description | #comments |
|---|---|
| **Expected features supported in REME-D at the time of the study** | |
| inspecting actor state | 5 |
| inspecting mailbox contents | 4 |
| breakpoints | 3 |
| step-by-step execution | 2 |
| pause command | 1 |
| **Expected features supported in current version of REME-D** | |
| inspecting state inside of an object | 3 |
| simulation of disconnected scenarios | 2 |
| message history | 2 |
| breakpoints extensions | 2 |
| decoupling of debugger core functionality from Eclipse IDE | 1 |
| **Expected features interesting for future research in REME-D** | |
| mapping breakpointed messages to lines of code | 1 |
| better visualization of actor view | 1 |
| adding message view | 1 |

Table 3: Summary of comments about expected features.

In 1993, Pancake and Netzer published one of the most relevant bibliographic efforts in the field including 293 entries [36]. This effort persisted in their online bibliography [37] which, on the last update in 1997, counted no less than 659 entries about technical reports, journal and conference papers, and Phd dissertations dealing with parallel and distributed debuggers. Many of those techniques are nowadays outdated because of the rapid advances in the latests years in both hardware and software [20], e.g., GUI-based front ends for debuggers is nowadays a given.

In this section, we highlight a number of distributed debuggers which have influenced the design of REME-D in significant ways. Since REME-D lies at the intersection of two families [3] of debuggers, namely breakpoint-based and event-based debuggers, we highlight related work in both domains. We also compare our approach with current proposals aimed at debugging or understanding web applications that rely on JavaScript. We deem these approaches related, as the execution model of AJAX-based applications strongly resembles that of distributed event-loops which REME-D targets.

*7.1. Breakpoint-based Debuggers*

Breakpoint-based debuggers execute the program in *debug* mode under the control of a debugger that allows programers to pause/resume program execution, inspect program state, and perform step-by-step execution. Most distributed breakpoint-based debuggers see as building blocks processes that communicate with each other by means of message passing. Essentially, each process is controlled by a sequential debugger, and coordination between them is carried out by a centralized console or GUI. Most-well known examples of these types of debuggers include research prototypes like p2d2 [7], Node Prism [9], Net-Dbx [10], and CDB [20], and commercial debuggers such as TotalView [8], IBM's Distributed Debugger [38], and Allinea DDT [39]. In this work, we also take a breakpoint-based approach as its features provide a simple but fundamental debugging toolbox.

One of the main critiques to the aforementioned debuggers is that the level of granularity of commands is limited and too-fine grained as the focus is on the source code, making debugging complicated and the amount of information overwhelming [14]. Millipede [14] aims to solve this by introducing a multi-level debugging approach which consists of three levels: the sequential level (controlling the intra-process execution), the message level (controlling messages interchanged between processes), and finally a protocol level (concerned with communication protocol). Message level breakpoints are a suitable breakpoint model for a non-blocking concurrency model, but they are rather low level abstractions since they only allow to stop and step the execution of one PVM API call. The message breakpoint proposed in REME-D has been inspired by Wismuller's *message breakpoints* [17]. In [17], a message breakpoint stops all receiver processes of the next message sent by a process. The combination of a message breakpoint with a traditional breakpoint on the send statements provides similar semantics to REME-D's step-into command. REME-D's breakpoint catalog transcends Wismuller's message breakpoints since it also provides breakpoint semantics for future type message passing.

Some of the breakpoint-based debuggers allow to set breakpoints on statements of one process (e.g., TotalView) or a set of processes (e.g., p2d2, Node Prism). Being able to set breakpoints on a set of process is specially interesting in the context of ambient-oriented applications since a number of devices share same source code. REME-D's support to specify on which device a breakpoint should be active has been inspired by p2d2.

The main feature lacking in current distributed debuggers to fit mobile networks is a means to deal with frequent disconnections. For the most part, dis-

tributed debuggers assume a stable network infrastructure. Fragile communication channels are assumed to be handled at the application level, i.e., communication failures are seen as an application-level errors. However, in a mobile setting, it is desirable that the debugger gracefully deal with network disconnections. A relevant exception is TotalView which supports open debugging sessions to some extent by relying on the underlying MPI middleware to manage and connect to new or independently started processes. This gives a degree of freedom in the configuration of a debugging session, making it attractive for an ambient-oriented debugger. However, in contrast to REME-D, the target application still needs to be compiled in a special way in order to be able to interact with TotalView's debugging agent before it can be dynamically included in the debugging session.

### *7.2. Event-based Debugging Tools*

Event-based debuggers [3] conceive the execution of a program as a sequence of "events". An event may be a MPI API call, read/write memory, send/receive functions, etc. Event-based debuggers record the trace of the events generated by the application (often called *event history*) during its execution. The event history can then be used to either browse the events once the application is finished [13, 16], or to replay the execution of a program in order to recreate the conditions under which the bug was observed [40, 15, 12, 41]. Analysis of this history varies from presenting the raw data to the user for inspection, to relying on graph-based analysis methods, or supporting graphical visualization techniques (e.g., time-space diagrams [42, 43], message and process order views [13]).

Event-based debuggers have been criticized mainly because of the overhead of collecting and saving information. Also browsing an event history does not scale since manually inspecting huge traces becomes cumbersome and difficult [3]. As such many research efforts have focused on reducing the amount of events recorded or presented to the user [23, 15, 44, 12]. Nevertheless, event-based debuggers fit well with a non-blocking concurrency model as message sends and receipts can be represented as separate events. A partial order of such events would accurately reflect the behaviour of a distributed application. Some approaches explore a partial order of the event history based on the *happened before* relation for browsing [13] or replay [15, 8, 7]. The happened-before relation shows how events potentially affect each other [19], allowing developers to identify potential places that caused a bug and as such, offering a similar functionality as stack traces in sequential debuggers. REME-D 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.

Within the field of event-based debugging tools, REME-D's closest work is Causeway [13], a message-oriented distributed debugger for the language E[6]. E is a distributed language designed for writing secure peer-to-peer distributed programs for open networks from which AmbientTalk's inherits its non-blocking concurrency model. In contrast to REME-D, Causeway is a post-mortem debugging tool. As such, programmers have to run over and over a program until the bug would appear, making the reproduction of a rare erroneous condition even rarer. Moreover, as remarked by Dao et al. in [45], reproducing the spectrum of possible states that a distributed application can be in and exposing the application to them before its deployment may be not feasible. This is more exacerbated in ambient-oriented application due to the dynamic environment in which they run. The open debugging support is thus essential feature that saves the programmer from having to restart the program to recreate the conditions of errors, since the debugger can attach to a running application and devices can be incorporated dynamically to a debugging session at runtime. This feature is unique to an ambient-oriented debugger, and not supported by Causeway.

### *7.3. Other Approaches*

A number of distributed debugging techniques have been proposed for programming models besides object-orientation, such as actors. IC2D [11] is a graphical environment for monitoring and managing distributed ProActive applications (running on a grid). In order to monitor ProActive computations, it provides graphical visualisation including views to visualize the topology of active objects, and message sends and receipts for selected active objects. These visualization is equivalent to the REME-D'S Debug and Actor State views described in section 4. IC2D also allows to interactively add a new or existing mobile active object to any running ProActive node as well as to move active objects to other nodes displayed by IC2D. However, in contrast to REME-D, unanticipated ProActive active objects cannot be added to existing debugging sessions.

*AJAX and JavaScript-based approaches.* Modern web applications rely on AJAX [46] and JavaScript to provide a high degree of interactivity with the user. These web applications move away from the traditional page-based navigation, into one in which the document rendered by the browser is changed on the fly as a response to asynchronous communications with the application server. Similarly to ambient-oriented applications, AJAX programs work as communicating event loops processing user interface events as well as asynchronous messages from a server.

While most tools that aid developers in debugging these applications (such as FireBug[6] and the Chrome DevTools[7]) concentrate on the client side behaviour alone, little attention has been payed at the debugging or comprehension of the interaction between the client and the server. A notable exception is FireDetective by Matthijssen et. al [47]. FireDetective traces the execution of both client (JavaScript) and Server (JavaEE) applications, and provides a unified visualization. In contrast to the previously discussed Causeway debugger, FireDetective is not a post-mortem debugger, but rather the traces are visualized as they occur. The main aim of FireDetective is the same as REME-D's: to provide insight into the causal relations (happens-before) hidden by the use of asynchronous message passing present in AJAX and AmbientTalk respectively. FireDetective, however, is not a debugger since it does not provide direct control over the execution of either client or server. Augmenting FireDetective with the debugging commands provided by REME-D would prove to be an interesting direction of future work.

While there exist some JavaScript libraries such as Q[8] that offer future-type message passing. To the best of our knowledge, no debugging support is provided for those libraries. As such, developers cannot place breakpoints on asynchronous messages that will stop when the future is resolved such as in REME-D. Stepping commands such as the step-return or step-into proposed in REME-D would ease the debugging of future-type message passing interactions in the web.

## 8. Discussion and Conclusion

In ambient-oriented programming, the complexity of programming in a distributed setting is married with the network fragility and open topology of mobile applications. Debugging under this conditions makes it so that existing approaches are insufficient, and a new kind of debugging, that we term *ambient-oriented debugging* is warranted. We identify two main challenges that ambient-oriented debugging must address: message-oriented debugging and open debug sessions. To address these challenges, we introduced an online ambient-oriented debugger called REME-D.

REME-D's principal contribution lies in that it implements the features of ambient-oriented debuggers as an ambient-oriented application which incorporates breakpoint-based debugging methodology where the focus is placed on the

[6] `http://getfirebug.com/`
[7] `https://developers.google.com/chrome-developer-tools/`
[8] `https://github.com/kriskowal/q`

exchange of asynchronous messages between actors. More concretely, REME-D adapts features from breakpoint-based debuggers to event loop concurrency — actor state inspection, message breakpoints, stepping over or into turns— , while incorporating for online usage features from post-mortem, message-oriented debuggers —browsing causal links.

REME-D proposes epidemic debugging as a mechanism to address the openness of AOD: it can install itself on newly discovered devices, a process in which REME-D spreads to devices joining the debugging session. Devices can leave the debugging session, either due to communication failures or in response to a user action, without disrupting the debugging of the remaining participants. REME-D implements those features by exploiting AmbientTalk's reflective API, resulting in a modular, reusable and flexible design that shows that it is possible to build tool support in tandem with the programming support for dealing with partial failures.

Considering the results of the user study, we distill three valuable insights. First, the features that participants actually expected from an ambient-oriented debugger were indeed supported in REME-D. This observation is based on the analysis of both the pretest statements and the suggestions that participants freely left on space provided for comments. Second, participants valued REME-D as a program understanding tool suited to make ambient-oriented programming in AmbientTalk easier. Finally, the Eclipse UI interface is relevant to how users perceive and value the features of REME-D, and it requires further attention.

We can foresee several avenues for future work. First, further effort must be spent on the UI of the prototype. The user study revealed that the UI components from the Eclipse Debug Plugin, designed for sequential debuggers, are not fit for an AOD. Instead, a custom-made UI which provides the user with graphical representations for AOD concepts (e.g., incoming and outgoing messages) is needed. Second, further support for causal link navigation should be implemented. So far, REME-D allows developers to see from where a message comes from, but not the state of the actor that sent it (at the moment it was sent). This is due to the decoupling in time between a message send and a message being processed. In order to address this, back-in-time debugger techniques, where the state of an actor is saved as part of the causal link of a message, should be explored. Finally, we would also like to develop a dedicated UI for the Android platform in order to explore "live debugging" of applications running on mobile devices, by permitting developers to control a debugging session, not from an IDE but from the device itself.

**Acknowledgements**

Elisa Gonzalez Boix and Carlos Noguera are funded by the MobiCraNT project of the Brussels Institute for Research and Innovation (Innoviris).

**References**

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

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

[3] C. E. Mcdowell, D. P. Helmbold, Debugging concurrent programs, ACM Computing Surveys 21 (1989) 593–622.

[4] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter, Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks, in: Inter. Conf. of the Chilean Computer Science Society (SCCC), IEEE Computer Society, 2007, pp. 3–12.

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

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

[7] R. Hood, The p2d2 project: building a portable distributed debugger, in: Proc. of the SIGMETRICS symposium on Parallel and distributed tools (SPDT), ACM, New York, NY, USA, 1996, pp. 127–136.

[8] C. Gottbrath, Deterministically Troubleshooting Network Applications, Technical Report, TotalView Technologies, 2009.

[9] S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, R. Title, A scalable debugger for massively parallel message-passing programs, IEEE Parallel Distrib. Technol. 2 (1994) 50–56.

[10] P. Neophytou, N. Neophytou, P. Evripidou, Debugging MPI grid applications using Net-dbx, in: European Across Grids Conference, Lecture Notes in Computer Science, pp. 139–148.

[11] F. Baude, A. Bergel, D. Caromel, F. Huet, O. Nano, J. Vayssière, Ic2d: Interactive control and debugging of distribution, in: Proceedings of the Third International Conference on Large-Scale Scientific Computing-Revised Papers, LSSC '01, Springer-Verlag, London, UK, UK, 2001, pp. 193–200.

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

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

[14] E. Tribou, J. Pedersen, Millipede: A multilevel debugging environment for distributed systems, in: Proc. of the Inter. Conf. on Parallel and Distributed Processing Techniques and Appl. (PDPTA), volume 1, Las Vegas Nevada, USA, pp. 187–193.

[15] R. H. B. Netzer, B. P. Miller, Optimal tracing and replay for debugging message-passing parallel programs, in: Supercomputing '92: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 502–511.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, I. Stoica, X-Trace: A pervasive network tracing framework, in: 4th USENIX Symposium on Networked Systems Design & Implementation, Cambridge MA, USA, pp. 271 – 284.

[17] R. Wismüller, Debugging message passing programs using invisible message tags, in: Proc. of the European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, 1997, pp. 295–302.

[18] G. Kola, T. Kosar, M. Livny, Faults in large distributed systems and what we can do about them, in: Proceedings of the 11th international Euro-Par conference on Parallel Processing, Euro-Par'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 442–453.

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

[20] X. Wu, Q. Chen, X.-H. Sun, Design and development of a scalable distributed debugger for cluster computing, Cluster Computing 5 (2002) 365–375.

[21] J. Quigley, The white programming language, 2007. CODE Group, Illinois Institute of Technology. `http://dijkstra.cs.iit.edu/code/white`.

[22] E. Bainomugisha, J. Vallejos, E. G. Boix, P. Costanza, T. D'Hondt, W. De Meuter, Bringing scheme programming to the iPhone Experience, Software: Practice and Experience 42 (2012) 331–356.

[23] G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, in: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 535–552.

[24] E. Gonzalez Boix, Handling Partial Failures in Mobile Ad hoc Network Applications: From Programming Language Design to Tool Support, Ph.D. thesis, Vrije Universiteit Brussel, Faculty of Sciences, Software Languages Lab, 2012.

[25] G. Kiczales, J. D. Rivieres, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.

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

[27] G. Marceau, G. Cooper, J. Spiro, S. Krishnamurthi, S. Reiss, The design and implementation of a dataflow language for scriptable debugging, Automated Software Engineering 14 (2007) 59–86.

[28] Y. P. Khoo, J. S. Foster, M. Hicks, Expositor: Scriptable time-travel debugging with first-class traces, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 352–361.

[29] P. C. Bates, Debugging heterogeneous distributed systems using event-based models of behavior, ACM Trans. Comput. Syst. 13 (1995) 1–31.

[30] R. A. Olsson, R. H. Crawford, W. W. Ho, A dataflow approach to event-based debugging, Software: Practice and Experience 21 (1991) 209–229.

[31] C. Zhang, D. Yan, J. Zhao, Y. Chen, S. Yang, Bpgen: an automated breakpoint generator for debugging, in: Software Engineering, 2010 ACM/IEEE 32nd International Conference on, volume 2, pp. 271–274.

[32] P. J. Astudillo, A Distributed Back-in-Time Debugger for Ambient-Oriented Programs, Master's thesis, Vrije Universiteit Brussels, Faculty of Sciences, Software Languages Lab, 2012.

[33] D. Campbell, J. Stanley, Experimental and Quasi-Experimental Designs for Research, Houghton Mifflin Company, 1963.

[34] M. K. A., Quasi-experimental evaluations. part 6 in a series on practical evaluation methods, Research-to-Results Brief (2008).

[35] V. B. Kampenes, T. Dybå, J. E. Hannay, D. I. K. Sjøberg, A systematic review of quasi-experiments in software engineering, Inf. Softw. Technol. 51 (2009) 71–82.

[36] C. M. Pancake, R. H. B. Netzer, A bibliography of parallel debuggers, 1993 edition, in: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging, PADD '93, ACM, New York, NY, USA, 1993, pp. 169–186.

[37] C. M. Pancake, R. H. B. Netzer, Bibliography on parallel and distributed debuggers, 2004. `http://liinwww.ira.uka.de/bibliography/Parallel/debug_3.1.html` (captured in June 2012).

[38] M. S. Meier, K. L. Miller, D. P. Pazel, J. R. Rao, J. R. Russell, Experiences with building distributed debuggers, in: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, SPDT '96, ACM, New York, NY, USA, 1996, pp. 70–79.

[39] A. DDT, The Distributed Debugging Tool, Technical Report, Allinea, 2012. `http://www.allinea.com/Portals/90122/docs/user-guides-and-technical-docs/allinea-ddt-3.1-user-guide-may-2012.pdf` (captured May 2012).

[40] N. Thoai, D. Kranzlmüller, J. Volkert, Shortcut replay: A replay technique for debugging long-running parallel programs, in: Proceedings of the7th Asian Computing Science Conference on Advances in Computing Science:

Internet Computing and Modeling, Grid Computing, Peer-to-Peer Computing, and Cluster, ASIAN '02, Springer-Verlag, London, UK, UK, 2002, pp. 34–46.

[41] T. J. LeBlanc, J. M. Mellor-Crummey, Debugging parallel programs with instant replay, IEEE Trans. Comput. 36 (1987) 471–482.

[42] M. Frumkin, R. Hood, L. Lopez, Trace-driven debugging of message passing programs, in: 12th International Parallel Processing Symposium, IPPS '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 753–762.

[43] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. Bruce, I. Karen, L. Karavanic, K. Kunchithapadam, T. Newhall, The paradyn parallel performance measurement tools, IEEE Computer (1995).

[44] M. RONSSE, D. KRANZLMULLER, Rolt(mp) - replay of lamport timestamps for message passing systems, PROCEEDINGS OF THE SIXTH EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING - PDP '98 (1998) 87–93.

[45] D. Dao, J. Albrecht, C. Killian, A. Vahdat, Live debugging of distributed systems, in: Proceedings of the 18th International Conference on Compiler Construction: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, CC '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 94–108.

[46] J. J. Garrett, Ajax: A new approach to web applications, `http://adaptivepath.com/ideas/essays/archives/000385.php`, 2005. [Online; Stand 18.03.2008].

[47] N. Matthijssen, A. Zaidman, M.-A. Storey, I. Bull, A. van Deursen, Connecting traces: Understanding client-server interactions in ajax applications, in: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 216–225.