The Journal of Systems and Software 90 (2014) 76–90

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Distributed debugging for mobile networks

Elisa Gonzalez Boix*, Carlos Noguera, Wolfgang De Meuter

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium

ARTICLE INFO

Article history:
Received 31 January 2013
Received in revised form 11 October 2013
Accepted 13 November 2013
Available online 4 December 2013

Keywords:
Distributed debugging
Distributed object-oriented applications
Mobile networks

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.

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 (Cheung et al., 1990). 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* (Gait, 1985; Mcdowell and Helmbold, 1989).

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 (Van Cutsem et al., 2007). 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 (Agha, 1986), event loop concurrency (Miller et al., 2005)).

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 (Hood, 1996), TotalView (Gottbrath, 2009), Node Prism (Sistare et al., 1994)), for grid computing (e.g., Net-Dbx (Neophytou et al., 2013), and IC2D (Baude et al., 2001)), or for general-purpose distributed computing in fixed, stationary networks (e.g., Amoeba (Elshoff, 1989), Causeway (Stanley et al., 2009), and Millipede (Tribou and Pedersen, 2013)). 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

* Corresponding author. Tel.: +32 2 6293581; fax: +32 2 6293525.
E-mail address: egonzale@vub.ac.be (E. Gonzalez Boix).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.11.1099

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 (Van Cutsem et al., 2007) (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 (Hood, 1996; Netzer and Miller, 1992; Fonseca et al., 2013; Stanley et al., 2009; Wismüller, 1997); 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 Stanley et al. (2009) 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.

Fig. 1 gives a graphical overview of the checkout protocol (verifying the aforementioned requirements) modeled 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`.

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 Fig. 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* (Kola et al., 2005) (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 (Lamport, 1978) between them. In sequential debugging, a