Comparison, Replay, and Refinement of Communication Traces for Debugging Distributed Failures

from

Tim Felgentreff

A thesis submitted to the
Hasso-Plattner-Institute for Software Systems Engineering
at the University of Potsdam, Germany
in partial fulfillment of the requirements for the degree of

Master of Science in Software Engineering

Supervisors

Prof. Dr. Robert Hirschfeld
Michael Perscheid

Software Architecture Group
Hasso-Plattner-Institute
University of Potsdam, Germany

August 20, 2015

Abstract

An increasing number of companies build their business on distributed Web applications. Hosting providers respond to that demand and made it easier to deploy systems that spread across multiple services. However, this trend has outpaced the development of adequate debugging tools and developers still have to rely on an improvised patchwork of symbolic debuggers and printf debugging to find failure causes.

Consequently, observing and debugging failures in distributed applications is laborious and time consuming. As multiple processes execute in parallel, developers cannot systematically stop and inspect the entire system. Although processes are implemented in different languages, and often tightly coupled, debugging tools that span across processes and languages are practically non-existent. Instead, developers have to use different debugging techniques to inspect each process sequentially. This makes it hard to reason about causality, because the network communication changes when debugging a process. That may lead to failures that occur in deployed systems, but that developers cannot reproduce locally.

This work presents an integrated method to support debugging of distributed systems in four steps: First, we continually trace network communication to have accurate data about event order if a failure occurs in a deployed system. Second, we compare communication patterns heuristically to identify anomalous differences and guide developers in search of failure causes. Third, we replay failing network schedules to reproduce failures in a local test environment. Finally, we refine the traced data during replays and connect network events to code locations. We present network events and code within one tool, so developers can use both levels of abstraction for debugging.

We show how our approach can be used to inspect failures in distributed systems that would be hard to debug with traditional approaches.

Zusammenfassung

Die Zahl der Firmen, die als Teil ihres Geschäftes verteilte Web Anwendungen entwickeln, nimmt zu. Anbieter zur Bereitstellung von Servern bedienen die steigende Nachfrage, und erleichtern den Einsatz von Systemen die sich über mehrere Dienste hinweg erstrecken. Dieser Trend jedoch schreitet schneller voran, als die Diagnoseprogramme für diesen Einsatzbereich weiterentwickelt werden, weshalb Entwickler noch immer mit symbolischen Fehlersuchwerkzeugen und Protokollierung arbeiten müssen, um Fehler zu finden.

Folglich ist das Beobachten und Beheben von Fehlern in verteilten Systemen noch immer mühselig und zeitaufwendig. Da mehrere Prozesse parallel laufen, können Entwickler nicht das gesamte System zur Untersuchung anhalten. Obwohl Prozesse in mehreren Sprachen implementiert, und eng gekoppelt sind, gibt es praktisch keine Werkzeuge, die über Prozess- und Sprachgrenzen hinweg funktionieren. Stattdessen müssen Entwickler verschiedene Fehlersuchtechniken für jeden Prozess einzeln anwenden. Das erschwert es ihnen, kausale Zusammenhänge zu verstehen, da sich das Kommunikationsverhalten ändert, wenn einzelne Prozesse untersucht werden. Das kann dazu führen, dass Fehler, die im Produktivsystem auftreten, von Entwicklern lokal nicht reproduziert werden können.

In dieser Arbeit wird eine Methode zur Fehlersuche in verteilten Systemen präsentiert, welche die folgenden vier Schritte umfasst: Zunächst wird fortwährend die Kommunikation im Netzwerk aufgezeichnet, um im Falle eines Fehlers im Produktivsystem den Entwicklern akkurate Ereignisdaten bereitzustellen. Daraufhin werden diese Daten mithilfe einer Heuristik auf Anomalien in den Kommunikationsmustern hin untersucht. Weiterhin werden diese Daten zur Wiedergabe der Kommunikation in einem Entwicklungssystem genutzt. Schließlich werden die aufgezeichneten Daten während der Wiedergaben verfeinert, und so die Netzwerkdaten mit dem Quelltext assoziiert. Beides wird in einem Werkzeug präsentiert, sodass Entwickler beide Abstraktionsebenen zur Fehlersuche nutzen können.

Wir zeigen wie unsere Methode genutzt werden kann, um Fehler in verteilten Systemen zu finden, die mit traditionellen Ansätzen nur schwierig untersucht werden können.

Contents

1 Introduction **1**
1.1 Approach . . . 3
1.2 Outline . . . 4

2 Request-based Distributed Applications **5**
2.1 Motivating Example . . . 6
2.1.1 System Description . . . 6
2.1.2 Failure Scenarios . . . 7
2.1.3 From Bug Report to Failure . . . 8
2.2 Challenges when Debugging in Distributed Environments . . . 10
2.2.1 Observation and Logging . . . 12
2.2.2 Understanding and Classifying Observations . . . 13
2.2.3 Reproducing Failures . . . 14
2.2.4 Connecting Server Events to Code Locations . . . 14

3 Replay-driven Fault Navigation **17**
3.1 Live System Observation . . . 17
3.1.1 Logical Communication Schedules . . . 18
3.1.2 Continuous Logging . . . 19
3.1.3 Mixed Environments . . . 20
3.2 Communication Analysis . . . 21
3.2.1 Differencing between Schedules . . . 21
3.2.2 Associating Variations and Assertions . . . 22
3.3 Replay and Reordering . . . 23
3.3.1 Re-ordering Scheduler . . . 24
3.3.2 Fault Navigation Over Multiple Runs . . . 25
3.3.3 Online Replay . . . 27
3.4 Debugging at Different Levels of Granularity . . . 30
3.4.1 Connecting Network Events to Code . . . 30

4 Implementation **33**
4.1 Tracing Communication Patterns . . . 34
4.1.1 Distributed Context-oriented Computing . . . 35

4.1.2 Tracer Entry Points . . . 36
4.1.3 Trace Format . . . 36
4.2 Communication Diffing . . . 36
4.2.1 Difference Format and Visualization . . . 37
4.2.2 Correlating Variances and Failures . . . 37
4.3 Reordering Access to Network Resources . . . 38
4.3.1 Constraints on the Networked Systems . . . 39
4.3.2 Entry Points . . . 39
4.3.3 Scheduling Execution . . . 40
4.3.4 Probable Replay . . . 41
4.4 Connecting Network and Object Communication . . . 41

5 Evaluation **43**
5.1 Tracing Overhead . . . 45
5.2 Case-Study: Reporting System . . . 47
5.3 Scheduling Accuracy . . . 49
5.4 Events to Code Mapping . . . 51

6 Related Work **53**
6.1 Record and Replay . . . 53
6.2 Replay-driven Fault Navigation . . . 55
6.3 Debugging at Different Levels of Abstraction . . . 55
6.4 Distributed Debugging . . . 57

7 Conclusion **59**

List of Figures

2.1 Flight Meta-Crawler Demo Setup 7
2.2 Flight Meta-Crawler Demo System 8

3.1 Activity Prior To Successful Diagnosis 22
3.2 Variations Failing Diagnosis Runs 23
3.3 FlowChart [10] of Re-ordering Scheduler 26

4.1 Interactions in the Prototype 33
4.2 Communication Schedule Overview 34
4.3 Captured Communication Logs 35
4.4 A Simple Differential Between 2 Schedules 37
4.5 Hue Mapping For Events 39
4.6 The Path Finder Network Extension 42

5.1 Reporting System 48

6.1 Causeway Message Debugger 56

List of Abbreviations

API	application programming interface
CLR	Common Language Runtime
COM	Component Object Model
COP	context-oriented programming
HTTP	the Hypertext Transfer Protocol
IAAS	Infrastructure-as-a-Service
IPC	Inter-process communication
JDI	Java Debugger Interface
JVMTI	Java Virtual Machine Tools Interface
JIT	just-in-time
JSON	JavaScript Object Notation
JVM	Java virtual machine
LCS	longest common sub-sequence
OSI	Open Systems Interconnection
PAAS	Platform-as-a-Service
REST	REpresentational State Transfer
SOA	Service-oriented Architectures
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
UUID	universally unique identifier
VCS	version control system
VM	virtual machine
XHR	XMLHttpRequest

1 Introduction

From its beginnings in the 1990s, the Web has become an important application platform in less than one and a half decades. Industry surveys indicate that an increasing number of companies develop distributed Web applications as a core part of their business [23, 14]. However, many of those applications are not written with distributed systems middleware to handle communication between services. Instead, developers improvise their own communication interfaces. This is error-prone and complicates integration [29].

Distributed applications are especially prone to software failures. Recent studies [6, 39] show a relative increase in code problems, hidden dependencies, and accidental complexity when integrating multiple systems. Different systems may provide different network interfaces and developers need to understand their utilization and implicit assumptions. Different systems also run different frameworks and programming languages, so that developers need help to associate observed network communication with code locations. This additional complexity makes it harder for developers to reason about the system as a whole, thus, they are more likely to introduce defects [42]. Debugging the resulting failures is hard, and we have identified four primary problems that developers face with current approaches and tools.

First, reliably reproducing failures is hard. Since distributed systems execute in parallel, a global event order is difficult to establish and may vary non-deterministically across executions [25]. Such variance depends on the physical connections between the servers. The development setup will differ from deployment and, thus, network delays and probabilities for non-deterministic behavior also differ. Consequently, developers may not be able to trigger some failures during development that occur in deployment, or such failures may occur so rarely that it is impossible to debug them effectively, because developers cannot reliably reproduce them.

Second, developers cannot easily distinguish between causes and effects. Many developers rely on systematic logging to understand failures that only occur during deployment [42]. However, isolated event logs are poorly suited to create a global view on the events in a distributed application. The ability to reason about global event order is necessary for humans to understand causality in a system [25]. Understanding the cause of a failure is essential for fixing it correctly.

Third, developers cannot easily inspect execution on servers in the system. With symbolic debuggers, programmers inspect execution state and behavior to find anomalies that may have caused a failure. Symbolic debuggers enable them to stop execution and interactively inspect the program's state. However, in a distributed system, developers need to inspect state across multiple servers, which execute in parallel. Symbolic debuggers are tailored towards systems under developer control, but not all parts of a distributed system provide that kind of control. Stopping servers with a debugger to inspect the execution state is not feasible during deployment and external services may not provide any debugging mechanism.

Finally, different frameworks and programming languages on several servers force developers to apply multiple debugging tools. Consequently, developers cannot debug and inspect the distributed system as one. Instead, they have to treat each system separately, because current debugging tools cannot connect and synchronize with others. Support for debuggers that work across multiple servers and languages is virtually non existent.

Due to these problems, debugging failures in distributed systems, especially non-deterministic failures, is a time-consuming task. It usually consists of aggregating various forms of log files, debug output, and application programming interface (API) documentation for the employed services, and then debugging on an equivalent, local system to infer possible problems. Symbolic debuggers cannot be used in distributed applications, because they only support independent analysis of individual processes and require full control over those.

Tool support for network observation, log aggregation and debugging across different systems is virtually non-existent. Yet, to improve programmer productivity, providing adequate debugging tools is a key ingredient, as without them, debugging is a manual, time-consuming task [12].

We argue that there is a need for debugging tools that target distributed, heterogeneous systems which are not entirely under developer control. Architectural patterns for the these kinds of systems, such as Service-oriented Architectures (SOA) and REpresentational State Transfer (REST); and hosting services, such as Platform-as-a-Service (PAAS) and Infrastructure-as-a-Service (IAAS), are relatively new phenomena—the first comprehensive description of SOA was in 2005 [15]. For that reason, much of the work in this area is still not aimed at the specific challenges of debugging distributed applications. Developers of these systems need tools that provide a means to record and replay events consistently across multiple servers. The tools should furthermore guide developers towards anomalous differences across multiple executions, so developers can reason about causality in the system. Finally, such a tool should allow developers to cope

with multiple frameworks and languages, as they are common in distributed applications.

1.1 Approach

In this work we present an approach for integrated debugging in distributed applications, targeted specifically at Web development, but extensible to other kinds of applications with similar problems. Our approach—Replay-driven Fault Navigation—combines state of the art techniques in the area of record and replay to eliminate of non-determinism, distributed program comprehension, and statistical debugging.

To allow debugging of non-deterministic failures, we present a tracing approach for unified monitoring of systems. We continuously monitor the live system and record enough data to allow reproducing relevant program state at a later time. Our monitoring imposes little overhead so it does not disturb execution unduly and can stay enabled during deployment. We analyze the recorded data and correlate it to failures to determine causes and effects. Network schedules are treated as sequences of send and receive events between pairs of servers. Our heuristics assign cause probabilities to sub-sequences of the recorded network schedules, to guide developers during debugging. Developers can replay such event sequences with our network scheduler and iteratively refine recorded execution data for inspection. The scheduler is an intermediate layer that controls how network events pass between the application code and the framework. It blocks or stalls events to change the order in which they pass in and out of the application to match the recorded sequence. We correlate the network data with source code and present both in a unified debugging tool, so developers can debug at either level of abstraction and correlate source code across servers and languages.

The advantages of our approach over traditional stop-the-world symbolic debuggers are: the impact on the runtime behavior is kept at a minimum, the technique can be applied to the live system, eliminating the need for a test setup, and the debugging tools can help developers to logically connect source code locations across different servers.

To our knowledge, the presented approach is unique in that it explicitly tries to support debugging across different languages and system with one integrated approach. Work on understanding and debugging distributed systems has been on-going for some time [30] and the state of the art has solutions for each part of Replay-driven Fault Navigation, but no integrated solution exists that we are aware of.

1.2 Outline

This work is structured as follows: in chapter 2, we discuss the problem domain on a small example. We highlight challenges faced by developers when working with current debugging techniques. In chapter 3, we present our four-step approach to those challenges, and how that approach could work in different languages and deployment scenarios. In chapter 4, we discuss an implementation of Replay-driven Fault Navigation in Smalltalk. Finally in chapter 5, we use the initial example and a larger application adapted from an industrial system to evaluate our tool in comparison with contemporary techniques. In chapter 6, we discuss related work and present our conclusions in chapter 7.

2 Request-based Distributed Applications

A distributed application consists of a number of processes running on one or more servers that are communicating over a network, for the purposes of this work, the Internet. Such applications are especially prone to software bugs [6]. Software bugs are defects in a program that developers introduce through an error in thinking. If the defective code is executed it causes an infection of the execution state. If this infection propagates to other states, it may eventually become visible in form of a failure [42]. In this work we focus on debugging distributed Web applications, but our approach is applicable to other distributed applications as well.

Web applications are applications accessible through a Web browser, with, at least, a client-side component for interaction and a server-side component for data storage. Clients are written in JavaScript to run in the browser, and the server components may use any programming language. Many Web applications use the Hypertext Transfer Protocol (HTTP) as transfer protocol, with event-loops to handle asynchronous communication. The reason for this is the single-threaded nature of JavaScript in modern browsers. In the absence of multiple threads or processes, event loops are the only option to process multiple network events in parallel on the client. However, developers increasingly employ event loops for server implementations as well. Event loops avoid spawning large numbers of threads or processes that are—due to the potentially high network latencies across wide-area network—waiting for IO most of the time. Web applications have at least a client and a server component, but both often communicate with other services. These, *asynchronous request-based, distributed, multi-language applications*, are the subset of applications this work is aimed at.

The communication patterns of many distributed systems on the Web can be described using Unified Modeling Language (UML) sequence diagrams [35]. In a sequence diagram, each server is an entity with a lifeline for request handling, and receives and sends asynchronous requests. Even though HTTP is a synchronous protocol, if we regard both requests and responses as asynchronous requests, we can treat servers that handle parallel request using any mechanism as a single thread with only asynchronous request-handling using event-loops.

2.1 Motivating Example

We now present an example system that uses asynchronous communication between multiple servers. Meta-crawlers are a type of Web search engine that connect to other search engines. Meta-crawlers combine information from other sources and provide a unified interface to interact with that information. For example, flight booking systems that offer multiple airlines connect to the systems of the different airlines and provide a unified search and booking interface to customers. Meta-crawlers such as these offer more than just search across multiple providers, they provide additional services for comparing prices, comparing seating options, or booking flights.

2.1.1 System Description

We have built a system that implements such a meta-crawler and is connected to a service providing fake airline data. It consists of a Web site that allows the user to search for and reserve flights from an origin to a destination on a specific date. The server serving the Web site communicates with a cache. That cache provides flight information and caches it for a certain amount of time. Such flight information includes flight numbers, origin and destination of flights, information about intermediate airports for a flight, as well as pricing and available seats. This cache, in turn, connects to a flight-data generator service that stands in for the airlines providing information about their scheduled flights. All communication is done via JavaScript Object Notation (JSON)[1]. The setup for this system is shown if Figure 2.1, with the arrows indicating data direction.

An intermediate query and cache service is an optimization that meta-crawlers use, which avoids having to query the data sources directly on each user request. Querying on each request is undesirable due to network latencies and because data providers usually limit the number of queries per time unit that meta-services can send to save bandwidth and processing time. However, caches introduce the problems of stale data and cache invalidation. Prices for seats on airplanes usually rise as the flight date approaches, and as increasingly fewer seats are available. Consequently, in order to create a reservation, flight booking systems either cannot guarantee the exact prices for their results, or they have to invalidate their caches and query for current data at the time of reservation.

[1] JavaScript Object Notation
http://www.ietf.org/rfc/rfc4627.txt
accessed July 25, 2012

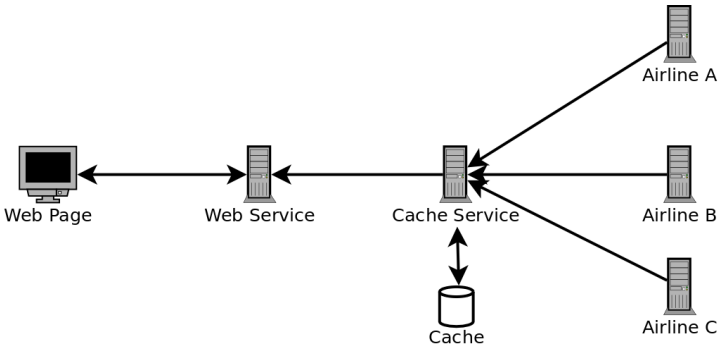

Figure 2.1: Flight Meta-Crawler Demo Setup

2.1.2 Failure Scenarios

Consider two customers on the meta-crawler Web page at the same time. Both are looking for flights from Berlin to Amsterdam, as shown in Figure 2.2. The results of their search will be served from the cache, and may not be the most recent data from the respective airlines. If both customers try to reserve seats on flight *LH1009* at roughly the same time, different things may happen.

First, depending on the geographical location of both users with respect to our system, the request for reservation may take some time to reach the server. Depending on the time difference, one reservation may complete entirely before the other request even arrives at the server. As we have said above, in order to create a guaranteed reservation, the Web server has to query the airline data—or, in our case, the fake airline data generator—for the most recent rates and available seats. Let us assume that the cache data was current and the first reservation request can be fulfilled at the displayed rate of 156 EUR. By the time the second request for reservation arrives at the Web server, the number of available seats has thus gone down to one, which means the prices has gone up as the airline adjusts pricing depending on available seats. Thus the second reservation can also complete, but only at a higher rate.

Second, it may be that both requests for reservation *arrive* at our server at the same time, and are served in parallel. This leads to a race condition for which we cannot deterministically decide which one of the users receives the lower rate and which one the higher rate. Either way, we, as the provider of the meta-service, do not want to end up reserving two seats and receiving only the lower rate from both customers, because in that case, depending on our and the airlines' terms of service, we may have to pay the price difference or a fine, which may not be a sustainable way to make business. At the same time, as a

Figure 2.2: Flight Meta-Crawler Demo System

customer, we do not want to create a binding reservation at any other rate than the one shown on the Web site at the time of our click on the *Reserve Seat* link.

There are a number of challenges for the developers of such a system. First, they have to understand the communication patterns that may arise in production, with parallel access to the Web site, cache, and airline data. They have to implement ways to deal with data races and non-determinism caused by network delays. To do that, they have to understand how non-determinism may lead to undesired states in their application, such as making a wrong reservation. To verify their solutions, they need a way to reliably reproduce problems and test them. Finally, they need to understand how communication between two systems tightly couples them at the implementation level, which means they have to understand how changes to the code in one system may impact that systems' ability to communicate with the others.

2.1.3 From Bug Report to Failure

A bug is an expression of a defect as a visible failure in the output of a system. To fix a bug, developers have to understand the failure, find the underlying defect, and remove it. Imagine the following bug report for our flight booking system:

> Sometimes, when I try to book a flight on your website, it is added to my cart just fine, but with a different price than what it was in the list when I clicked it.

This particular bug report is at a high abstraction level, as it describes only user-facing input and effects. For such bug reports, the first step towards fixing them is to reproduce them. Second, developers should write a minimal piece of executable code that reproduces the failure, so they can easily check their

attempts at a fix. Then the actual process of debugging can begin, which will eventually make the test case pass.

These steps can be time-consuming, especially for very abstract bug reports. Tools such as *ReCrash* [5] attempt to generate test cases automatically, but can only do so for deterministic failures. The above bug report contains the word "sometimes", indicating a degree of non-determinism. That non-determinism may have a number of causes. In this case, it may be that the user attempts to book flights slightly differently without noticing. There may be conditions in the environment, from browser bugs to transient hardware failures, that cause the issue to only appear sporadically. Or there may be non-determinism in the Web server implementation or the network communication.

To approach non-determinism during debugging, developers have to check each possible cause and try to eliminate it. Most of these causes, including network communication, are difficult to reproduce faithfully outside the live system. Unit tests are likely to be too low-level to expose complex interactions, and acceptance tests running in a test setup experience different conditions if the test setup is not similar enough to the live system. Developers cannot with certainty know whether a test setup sufficiently models the live system to reproduce a given failure.

Given the above bug report and its possible causes, developers at some point have to inspect the network communication schedule if no other cause can be found.

Scripted Diagnoses Instead of acceptance or unit tests in controlled test environments, developers may employ *system diagnoses* as "test cases" to simulate user input against the live system. Oftentimes those are simple scripts simulating a browser's communication with the Web page. This enables developers to include the circumstances under which the bug occurred in the test. Even if the bug only occurs "sometimes", repeated execution of the system diagnosis will eventually reproduce it if it is related to varying network schedules.

This is the approach taken by tools such as *iDNA* [8] and statistical debugging approaches. The deployed application is observed over long periods of time, possibly through multiple occurrences of the failure. If the failure occurs, as established by the bug report, "sometimes", then the diagnosis is good enough for Replay-driven Fault Navigation.

2.2 Challenges when Debugging in Distributed Environments

In distributed systems such as our flight booking system, developers face a set of challenges in addition to the challenges of non-distributed systems. The implementation of each service has to account for the unreliable network, developers need to understand the services they communicate with, and how data is represented and shared between those services.

Non-determinism Non-determinism may have different causes: from randomness in the implementation, varying shared resource access order, to interactions with other processes and the operating system schedule and transient faults in hardware. Non-determinism here means that for equal inputs, the output and execution speed of a program is indeterminate.
For a distributed system, non-determinism may mean that for an equal number and order of messages sent from one server, another server may produce different responses. A cause unique to distributed systems for this is network latency and request re-ordering. Sending a number of HTTP requests in one particular order does not guarantee its delivery in the same order. This is an instance of a leaky abstraction. HTTP messages may be broken up into multiple Transmission Control Protocol (TCP) packets. TCP packets may arrive out of order, and are collected by the operating system and delivered to applications in the original order for a connection. If the TCP packets for one HTTP connection are available, the operating system passes those packets to the application, which handles them as HTTP request. The application does not know that another HTTP request is on its way that was sent earlier from the point of view of the sender. So, if the processing of any request is influenced by any other, network latency may introduce non-determinism.
Network latency varies across network segments. Usually, during development, distributed systems are deployed in a test environment in an attempt to simulate the distribution properties of the deployed system. However, actual network latency during deployment is hard to predict, and data races across the network will only randomly be reproduced in a test environment. Furthermore, during development, *click-races*—the situation when two or more users try access a resource simultaneously—can oftentimes not be adequately tested. Finally, deployed distributed systems, due to ongoing communication, very rarely enter a globally consistent state, so proving a system correct for its projected uptime is difficult [17]. The only predictable way to enter consistent

states is to shut down and restart the whole system regularly, which is not desirable for many systems.

Heisenbugs Failures that occur only during or change during debugging are called *Heisenbugs* [18]. They happen because the debugging tool, through its observation process, changes the runtime behavior of the observed program. For non-deterministic failures, this means that introducing logging into the system may cause different failures to happen, or the original failure could happen much more or much less frequently, or not at all. This makes debugging non-deterministic failures harder, because the developer cannot be sure that the observed failure is the same as the original one.

Distributed systems exhibit Heisenbugs more often, because network access order varies even from slight disturbances, such as the operating system scheduler or other processes in the system. Most debuggers incur significant overhead [11], which causes equally great disturbances to network access order, which in turn may cause Heisenbugs [16].

Languages When systems run on different machines, or even only in different processes, and they communicate not via the language calling mechanisms, but rather through foreign-function interfaces, shared memory, or network communication, connecting call trees in one language to another is a non-trivial task.

A developer has to find exit and entry points for Inter-process communication (IPC) in all communicating systems, and relate the input and output data, to connect events within one system to another. This task is made more difficult by abstractions from the input and output, for example, while the operating system will deliver TCP packets in order to a language runtime, a Web framework will assemble HTTP messages before passing the data on to the application. If systems use different abstractions, one working on the transport layer and the other on the application layer, it is hard to determine the corresponding communication entry points.

Data Representation Different languages represent data differently. Consequently, different kinds of persistent data representation come natural to the language users. Different data representations have to be converted into each other, which introduces additional complexity and another source for bugs. Because the connection between the different systems and the data conversion is usually hidden behind libraries and frameworks, failures that are caused by incorrect data exchange are hard to debug, because developers have to understand multiple systems and how they transform the data.

For example, a web-page will naturally display information to a user in a visual manner, using nested HTML-DOM elements to express the information content in a user-friendly way. The JavaScript code running in the browser will

usually, at least intermittently, store information as JavaScript objects or in JSON. JSON may also be the format used in HTTP requests to transfer data between the client and the server. On the server, the data may again be transformed any number of times between different runtimes, backend servers, and databases.

These properties make debugging distributed systems a laborious task. We have identified the following four debugging activities to be especially problematic and in need of a solution.

2.2.1 Observation and Logging

The need for observation and logging of program state changes has clearly been recognized in both research and industry [36, 1, 37]. Most Web frameworks include logging facilities, different log-levels, and separate logs for access, input, output, erroneous and successful events. There are full libraries[2] and frameworks[3] that provide comprehensive logging facilities to applications.

Distributed system are setup differently during development than at deployment time. Simulation of distributed systems or parts thereof in small-scale development deployments, as used at development time of such systems, allows developers to evaluate system behavior in a simple setting, relative to the deployment setup. In such development setups, developers have full control over each server, and the power to stop and inspect each process at practically any point in the execution of the distributed system. Apart from such small-scale setups, developers use full-scale test deployments to provide a more realistic setting for stress testing critical aspects of the system. Using setups of varying degree between fully local deployments and full-scale test environments, developers can control the amount of realism their code is exposed to.

However, once deployed, distributed systems will over time encounter situations that could not have been anticipated by the developers, state changes accumulate, the underlying network topology changes, and, for successful services, system load and parallelism increases beyond what could reasonably be tested [17]. Long-running applications may experience non-deterministic, low-probability failures that develop over time. Thus, logging data from the deployed system is indispensable for testing and debugging failures that occur during deployment.

[2] Airbrake: The error app `https://airbrake.io/pages/home` accessed June 25, 2012

[3] Apache log4j Logging Services `http://logging.apache.org/log4j/1.2/index.html` accessed June 25, 2012

If problems occur, most logging frameworks have settings on how to react, by sending emails, checkpointing the running application, stopping and saving a stack-dump of the program state, etc. *heartbeat* services regularly check accessibility of services, existence of error artifacts, or output of logging frameworks, to filter and notify developers when a problem occurs in the deployed system.

Logging frameworks and error artifacts are useful for debugging, as long as only a limited number of frameworks is used. As long as logs are similar in format and level of abstraction, they can be related to one another. However, although most frameworks that are used in the context of networked applications offer logging capabilities, there is no widely accepted standard for logging information and formats across languages or frameworks.

2.2.2 Understanding and Classifying Observations

Given appropriate logging, developers have to review and understand the logs to determine how to approach a particular bug. This entails associating parts of a log or particular log entries with failures. Even for stack traces of single process applications, this is no easy task, and developers can easily get confused about where to look first in a trace [19]. For Web applications developers have to understand and associate parts of logs from different systems with the same failures. As different logs may expose different aspects of the same problem, they have to be aligned with respect to their order, so developers have chain of anomalous log entries to follow backwards. Without such alignment, developers can only guess which anomaly occurred first, and finding the cause of a failure is a matter of trial and error. But aligning multiple logs is a difficult task, unless there is a connected logging facility. Minimal clock skew across systems may render timestamps unusable for log ordering and parallel events cannot easily be recognized as such.

Another problem presents itself when the failure is logged at a different level of abstraction than the anomalies leading up to it. For example, if the log entry with HTTP status code 500 on the server represents the visible failure, the anomalies leading up to it may be in form of database access or JavaScript exceptions, but there may be multiple database accesses or exceptions for only one HTTP request. Developers have to connect these anomalies and group them at the right level of abstractions. Thus, understanding which log entries are relevant to a particular failure is hard to determine as well.

2.2.3 Reproducing Failures

Many debugging guides emphasize that the ability to reliably reproduce a failure is crucial in the attempt to fix the underlying defect, because it is the only way to make sure that an attempted fix is successful [28, 42, 19]. This can be difficult for single-process applications, however, distributed applications exacerbate this problem. The *global state* in a distributed system is hardly ever consistent, unless the whole system is restarted. That means that there is little opportunity for checkpointing the system for running through a failure from the checkpoint. To checkpoint a process, it needs to be in a consistent state, so it can restarted later. For multiple processes, this means that all nodes—and thus the complete distributed system—have to be in a well-known state before a checkpoint.

Provided a consistent checkpoint can be established *before* a failure occurs, reproduction may still fail. Non-deterministic failures caused by network latency or transient inconsistency in the global state will not manifest each time the defective code is triggered.

To deal with this problem, there are approaches to log HTTP traffic on-demand, when a failure occurs [24]. During replay, instead of just running the complete, distributed system, only one node is actually run, and interaction with external services is replayed from records.

This is useful if the developer knows on which node the defect is most likely to find. If that is unclear, however, the developer can only replay and debug on each node, sequentially, until the defect is found. For large numbers of communicating nodes, or if the error is not on any node the developer can access, this is a time-consuming process that may not lead to success.

2.2.4 Connecting Server Events to Code Locations

Provided a useful log and a replay mechanism exist, and developers have identified server events and servers that contribute to a failure, they have to change the level abstraction from network communication and servers to communicating processes and source code to find and fix the defect. In distributed systems, it is not possible to set a *breakpoint* on a network event, and stop all services that are communicating at the time to inspect them.

In order to deal with this problem, developers use logging in different ways. Simple logging mechanisms or frameworks can be used to determine the executed code around a particular network event. This increased logging can be enabled on-demand, and log, for example, method activations, arguments, and the data that is sent and received over the network [5]. Another approach is

to save a full continuation of the running program when a particular event occurs [13]. This continuation can later be inspected with a symbolic debugger.

The problem with these approaches is that they only provide a one-to-one connection between a network event and a code location. Yet, code locations are connected across servers via the network, so multiple code locations contribute to the same request. Developers need to identify the same network events as they are logged on different servers, to find those connections, and the debugging tools have to provide this identification.

Working with Multiple Call Trees

Debugging across communicating processes is similar to debugging across multiple threads. First, in both cases it is impossible to stop all execution in one atomic step. Second, all processes and threads have a separate stack as well. However, in contrast to multi-threaded applications, distributed applications cannot communicate through shared memory and have to use an IPC mechanisms instead. In the case of Web applications, HTTP APIs are the IPC mechanism.

Consider again the sequence diagram for expressing the runtime behavior of distributed systems. As per the UML standard, nested calls can be expressed using nested lifelines. Generally, that would be the case if a method call from one object to another causes the receiver to call another method on itself or a third object. For distributed applications, this can describe the relation of network messages and call trees. An HTTP request will trigger some code to be executed on the receiver. While the level of abstraction is different, in effect, this is equal to communicating threads and their interleaved call trees.

Each node executes code in one language, on one or more threads. The language runtime takes care of delivering calls to and return values from call-sites. On the network, the HTTP layer has that responsibility. This means, that in distributed systems there are multiple, nested layers of *runtimes*, each with their own call mechanism.

Developers of multi-threaded programs need to be able to move between call trees freely. The same applies for developers of distributed systems. While thread call trees within one runtime are at the same level of abstraction, network call trees and different language runtimes are not. There is very little support for connecting those layers within debugging tools, and methods for doing so are still being researched [37, 17].

3 Replay-driven Fault Navigation

With Replay-driven Fault Navigation, we address four challenges when debugging distributed applications: observing and logging failures, classifying and connecting logs, reproducing failures, and associating server events with code locations. More specifically, our approach consists of the following four steps:

1. We employ a lightweight record or logging mechanism on each server, to collect information about communication in the network. We combine the separate logs into one, by means of partial ordering using Lamport clocks [25]. This step provides an improvement over the heterogeneous and unordered logs developers have to deal with today.
2. We identify failing and passing runs among logged events by connecting assertion failures in the execution with observed network traffic. We compare failures and passes to infer critical regions in the systems' communication patterns. Using this information, developers can focus on the relevant sections of the logs.
3. We use recorded communication as patterns to replay failing network schedules and check the replays for assertion failures. If the defect is related to a network scheduling problem, the failure will reproduce reliably. In that case, developers gain confidence that the failure is indeed caused by variance in the network communication schedules. Additionally, developers gain a means to reliably reproduce the failure for further debugging.
4. Finally, we use failure replay for debugging. We refine recorded data during replay to associate network events with code locations. We present both in a tool that allows developers to move between the two abstractions freely [37]. This way, developers can inspect network events presumably corresponding to infected state at the program code level, an use additional replays to inspect execution state at this point.

3.1 Live System Observation

In order to understand a bug, developers iterate over the same code paths, inspecting different aspects of the program state. In order to do so, they need some notion of logical event order, to infer causal relations between events. In subsection 2.2.1, we also argued that it is important to have data from the live

system to debug distributed applications, because only the live system may exhibit the specific timing behavior to produce a sporadic bug.

3.1.1 Logical Communication Schedules

Communication schedules in distributed systems can only be established for subsets of all gathered network events, because communication between servers in a distributed system is truly parallel, and some events may thus happen at the same time, or so close in time as to have no effect on one another. To understand causes and effects in a distributed system, we have to establish the order of events in the communication.

Network communication events occur at different levels of abstraction, corresponding to layers of the Open Systems Interconnection (OSI) model [43]. We chose to establish schedules at the *application layer* of the OSI stack, so they may be easily correlated to the code that created them. Previous work has mostly focused on recording events on the *transport layer*, i.e. TCP and User Datagram Protocol (UDP) events [17, 24]. However, network and Web development frameworks usually work at the application layer, sending, for example, HTTP requests or SQL queries. These high-level events may correspond to multiple events on the transport layer. These transport events have little meaning for a developer working on the application layer, but combining the transport events back into application layer events is very difficult in a user space library that does not have access to e.g., TCP sequence numbers. Recoding schedules at the application layer allows us to provide a user-space solution at a, for developers, meaningful level of abstraction.

Partial Ordering of Events

In most systems, the order in which events occur is fundamentally important to our way of reasoning [25]. We say that event A happened at 3:15 if it happened *after* our clock read 3:15 and *before* it read 3:16. We call this order the *logical schedule* [24] in which the events occurred. The actual timing information (whether the event happened *at* 3:15, or between 1 and 59 seconds later), the *physical schedule*, is only a more specific representation of that ordering.

However, for systems capable of parallel execution, such as the distributed systems, a universal before relationship is not easily obtainable and may not physically exist. To solve this problem, logical, or *Lamport* clocks [25] are used to establish a partial ordering. Lamport clocks provide a logical schedule that can be implemented in a parallel system. Central to logical schedules is the causality

between events. If we know that an event A triggered an event B, we know that A must have happened before B, even in the absence of a clock.

The original implementation idea by Lamport, also implemented in this work, is still widely used [24, 37, 17, 1] and extension such as Vector Clocks [26], Version Vectors [30], and Interval Tree Clocks [31] exist for more specific scenarios.

3.1.2 Continuous Logging

The traditional stop-the-world debugging approach is ill-suited for a live system that only exhibits the erroneous behavior sporadically. We have argued in subsection 2.2.4 that developers cannot stop processes on different machines at the same time. First, this is technically impossible, because there is no way to perfectly synchronize all processes. Second, it is also impractical, as stopping the live system will make it unavailable for users.

Other debugging approaches observe the live system and gather enough data to allow inspection of the relevant program state at a later time. One approach is to log all execution state [37], so developers can inspect that information offline at a later time. However, such logging imposes a significant overhead, which makes this approach unsuitable for continuous logging in the live system. Other approaches [24, 17] have shown that it is possible to record just enough information to guide the system down the right path during subsequent executions. These latter approaches have little overhead and are suited for continuous logging.

We assume that each server in the network is in itself deterministic, and that any non-determinism is introduced by access-races to shared resources, i.e. the network and other servers. If each node is indeed deterministic, that means that if network requests arrive in the same order, the same *class* of output—correct or incorrect—is produced. Furthermore, we assume that nodes and the network are well-behaved, meaning, that nodes do not suddenly change their IP addresses, and network segments do not suddenly disappear, making nodes unreachable. We realize that this may preclude usage of our approach for less dependable networks.

This means we only have to record the order and type of HTTP requests between any two communication partners to faithfully reproduce communication, applying only traffic shaping. Compared to recording the request contents or partial execution traces, this is little information, with little space overhead. Indeed, [17] has shown that the necessary information overhead encoded into the request packages may be as little as adding an 8 byte Lamport clock. Since we do not store the message contents, these 8 byte plus identifiers for the two communication partners are all our storage requirements for network events at runtime. As identifiers, IP addresses and port numbers may suffice, adding

4 byte for the port and 16 byte for the address, in the case of IPv6 addressing. For an application that keeps the last 1,000,000 requests stored in its logs before rotation, this costs about 26 MiB in storage space.

Adding logging to an existing application will change its execution behavior. The additional execution steps impact performance and thus timing and memory behavior. Even slight changes can have a large impact if they increase the likelihood of, for example, cache misses in the CPU.

Nevertheless, choosing a lightweight logging mechanism is worthwhile to minimize disturbance. Furthermore, by enabling the logging facility in the deployed system, the overhead of logging is no longer a sporadic disturbance that only occurs when running a system diagnosis. Instead, the disturbed behavior becomes the normal behavior.

Minimizing disturbance for debugging is particularly useful for non-deterministic failures, but is a generally desirable property of debuggers. It minimizes the potential for failures that occur only during debugging, so-called *Heisenbugs*.

3.1.3 Mixed Environments

With continuous logging in a distributed system one has to consider organizational and practical boundaries. To introduce generalized logging into a distributed system we need to extend services on different nodes. Not all these nodes are under developer control, or have source code readily available to extend them. Our tool thus has to function in a mixed environment. For HTTP communication, this is easily possible by adding custom headers, which will be ignored by non-logging clients. We do not regard lower levels of communication. Ways to deal with services at the transport level have been covered in other work [17, 24].

Servers that are part of our distributed system, in the sense that communication with them contributes to the purpose of the system, can be either cooperating or non-cooperating. During normal execution, non-cooperating servers simply ignore the additional headers for Lamport clocks. To combine the logs at a later time, any discovery mechanism, from a well-known port to ZeroConf[1] can be used. Incoming and outgoing events for sent and received requests, respectively, can then be added for non-cooperating servers by means of inference.

This will necessarily create incomplete logs in the presence of non-cooperating servers, but this can be safely ignored if no state is shared with other, third-party, distributed systems accessing the same non-cooperating servers. These systems

[1] ZeroConf Internet Engineering Task Force (IETF) Internet Draft
`http://www.watersprings.org/pub/id/draft-ietf-zeroconf-reqts-12.txt`
accessed July 21, 2012

are common in practice, because REST and SOA encourage cooperating systems without shared state. However, this also means that our approach is not well suited for shared, "cloud" database services and other stateful systems.

3.2 Communication Analysis

In Replay-driven Fault Navigation, recording communication events provides data from the live system. Continuous logging will capture correct and anomalous communication events. Developers need to understand the difference, why one particular log ends with a failure, where another ends without one. There is little tool support to find such variances in server logs, consequently, this is the next step in our approach.

3.2.1 Differencing between Schedules

To create a tool to find variances in communication schedules, we need a model of such schedules. We argued in chapter 2 that sequence diagrams are well suited to both model network communication as well message passing between objects. Consequently, network schedules also form a *call tree*, with requests and responses representing message sends and returns.

However, network events occur at the higher abstraction level than message exchange between objects, and differences to single-threaded objects. In distributed systems, a single server serves more than one request in parallel, using event loops or multi-threading to process and send out responses as they become ready. Thus, we cannot assume a causal relationship between two requests arriving at a server one after another, and not between two responses leaving the server. The temporal relationship alone does not imply a causal relationship.

For non-deterministic failures caused by network schedules, however, we know that variances in the order of events influence one another. We can use our a diagnosis script to create multiple schedules that all yield the same result, but vary slightly. Comparing these schedules, we can find possible causal relationships between events. For sub-sequences of events that vary frequently across executions, these events can be assumed to have little or no influence on one another. Sub-sequences that vary only rarely or not at all are indicative of causal relations between those events.

Consider our flight meta-crawler bug report in section 2.1. We can assume that the discrepancy between the price displayed before and after reservation is due to a data race between another user reserving a seat on the same plane, and the user filing the bug report. If we run a system diagnosis it may pick up a number of users using the system, leading to activity as in Figure 3.1.

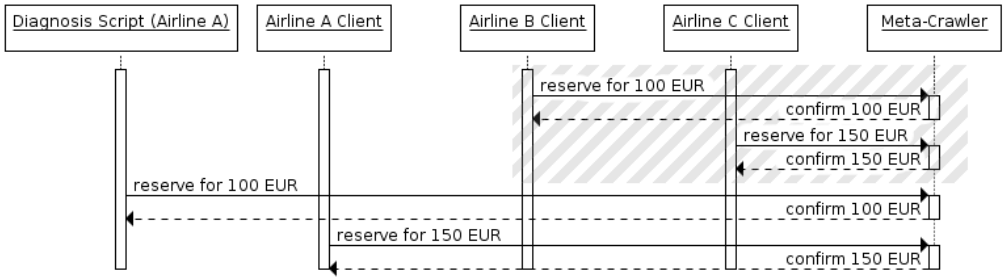

Figure 3.1: Activity Prior To Successful Diagnosis

Users booking other airlines prior to the diagnosis may reserve other flights without causing the diagnosis to fail. A variance in the reservations prior to the actual diagnosis—in the shaded area—is thus inconsequential to the result of the diagnosis. Only variations in the order of clients booking the same flight as tested in our diagnosis script have an impact on the result.

3.2.2 Associating Variations and Assertions

The diagnosis script is useful not only for producing similar schedules, it also includes assertions that test the outcome of the tested behavior. If the diagnosis reproduces a bug, those assertions will fail.

Variances between passing and failing schedules can be used to reduce the entry points that need to be checked for defects. If we compare all passing schedules, we find variances that probably have no bearing on the outcome of the test. If we compare them against the failing schedules, and subtract the variances within the passes themselves, we may find sequences that vary only between the passing and the failing schedules. Those may be of particular interest to the developer, because they are likely to be related to the failure. Such a variance can occur in the unshaded activity in Figure 3.1. If the second reservation for Airline A were sent prior to the receipt of the confirmation in the diagnosis script, a critical race may ensue that causes the diagnosis to fail.

Failing schedules have variances within themselves. We use this to heuristically mark these variances as less likely causes than others, as these variances then seem to have no impact on the outcome, since they all produce the failure. Variances within the failure schedules that also vary against all of the passing schedules are more likely to be only on the path of the infected state of the program, not the cause of the defect itself.

Consider Figure 3.2, as an example for variances in failing schedules. The failures, as the passes, may vary in the order of reservations prior to the actual diagnosed reservation—the shaded area near the top. However, the failures may

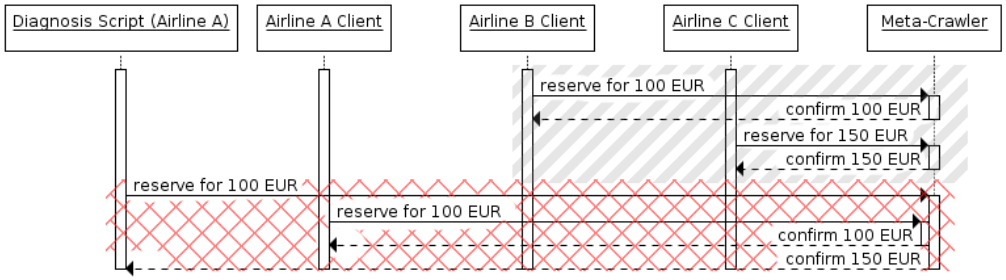

Figure 3.2: Variations Failing Diagnosis Runs

vary also in the order of the requests and confirmations for reservations made by the diagnosis and the Airline A client—the crossed area below. The only passing schedule for that part of the communication is the one we see in Figure 3.1. Thus, the initial variances within the failing schedule that also occur in the passes are of little interest to the developer and need not be considered. The variances near the end of the schedule, however, that are only between failing schedules and do not vary within the passes can be considered likely causes for the failure. We need to guide developers towards that part of the communication for further debugging.

Finally events that occur late in the schedule are less likely to be the original cause of a failure. To express this in our heuristic, we adjust the probabilities linearly with respect to the sequence number of the event within the schedule.

3.3 Replay and Reordering

Up to this point we have been inspecting the system without knowing whether varying communication schedules are the cause for a non-deterministic failure. We can now reinforce this assumption by running failing schedules multiple times through what we call our *re-ordering replay scheduler*. The scheduler enforces a particular order in network communication traces, which means we can replay a failing trace multiple times with exactly the same order of network events. If the result is consistently the same, even if the failure occurred only sporadically before, we have a strong case to believe that it was caused by this particular schedule, and further inquiry with our method is warranted.

In some cases the outcome may vary even if our scheduler enforces a failure schedule. An example is an improper implementation of timestamp parsing on one machine, that causes the bug whenever the timestamp has one of a few particular value. Such a bug will occur regardless of the network schedule, and our approach is not suitable to debug these kinds of failures. However, even in

that case the developer will have learned that non-determinism in the network schedules is not responsible for the failure, and can focus his debugging efforts on each server in turn, using traditional debugging approaches.

3.3.1 Re-ordering Scheduler

The scheduler itself works as an intermediate layer between the application code and the network, and effectively acts as a traffic shaper[2]. It tries to re-order access to the shared resource—the network—to enforce a particular, prerecorded schedule. This is done by holding and releasing incoming and outgoing communication according to the schedule, rather than the order in which they arrive.

Since we want to record application layer events, it is most natural to insert the scheduler between the application framework code responsible for sending and receiving, and the underlying language or operating system primitives. This has the additional advantage that the scheduler can be programmed with some knowledge about the framework, and thus the ability to communicate its actions to the developer on the level of abstraction of the framework. This not only keeps the developer in control of the debugging process, it also allows meaningful configuration of the scheduler heuristics. Because we record only the partial order and identifiers for communication partners for events, the scheduler can only replay scheduled events that occur in the system at the time of re-scheduling. That means that *all* servers that were active during record must be active during replay as well. This may not always be the case, and scheduling on the application layer allows us to communicate conditions for holding and releasing events to the developers, so they can take appropriate action, such as choosing another schedule or ensuring the required servers are indeed online.

As an example, the scheduler can to re-order according to a schedule that includes communication with a backup server. This backup server may only be active at particular times of the day, say, at night. Developers debugging a problem during the day may cause the scheduler to wait for a long time—up until the backup server is active. The system will seem to hang. Being able to communicate to the developers the reason for waiting helps them to understand the possible cause of the failure, and either bring the backup server online for the purposes of debugging, or choose a schedule recorded at another time.

Generally, such a scheduler has to control all points in a given framework where application layer requests are received and released. A possible algorithm for such a scheduler is given in Figure 3.3. Such a scheduler will have to dis-

[2] Internet standard definition of "Shaper" in RFC 2475
http://tools.ietf.org/html/rfc2475#section-2.3.3.3
accessed July 25, 2012

tinguish whether the framework is using multiple threads of execution or just one.

For our assumption of determinism on each server to hold, multi-threaded servers need to be synchronized on send and receive events, by using some kind of synchronization mechanism such as, for example, semaphores. When a request is received, the scheduler checks whether all requests that are expected to arrive before this request have been received. If they have, the requests can be released into the framework. Otherwise, the current thread of execution has to be suspended until all previously scheduled requests have been received. Upon release of a request into the framework, all threads waiting on this request can be released, too. Similarly, for sending responses, the scheduler holds those responses until all previously scheduled responses have been sent by the framework.

Single threaded servers as they are used, for example, for event loops, cannot be stopped if a request arrives too early, as that would prevent any other requests from being processed. In this case, the scheduler will receive requests into a buffer, and keep receiving until the currently scheduled requests arrives, which is then returned. Once the server is ready to process the next request, it may be released directly from the buffer, if it was previously received, skipping the network access altogether. In single-threaded mode we can ignore outgoing requests. Under the assumption that the server works deterministically, it will send the responses in the correct order if it receives all requests in the correct order.

3.3.2 Fault Navigation Over Multiple Runs

After reproducing a failure, developers have to inspect the program state. In section 3.1 we have argued that a lightweight logging approach is preferable for observations on the live system, both to minimize the performance impact and the possibility for Heisenbugs. However, this approach prohibits recording information about the program state at this stage.

With the scheduler, however, we can enforce failure schedules as needed for further analysis. Based on the initial communication log we can, in a re-run, add an initial, shallow log of the called methods on each node. This is based on the earlier work about incremental, online analysis of runtime behavior [33]. Given the source code running on the servers, this information can be connected in a debugging tool to present a call tree to the developers.

While developers explore the call tree, it is interactively refined on-demand. We add one-shot logging at points of interest as needed, without disturbing the deployed system unduly. Because the the scheduler holds on to parallel requests,

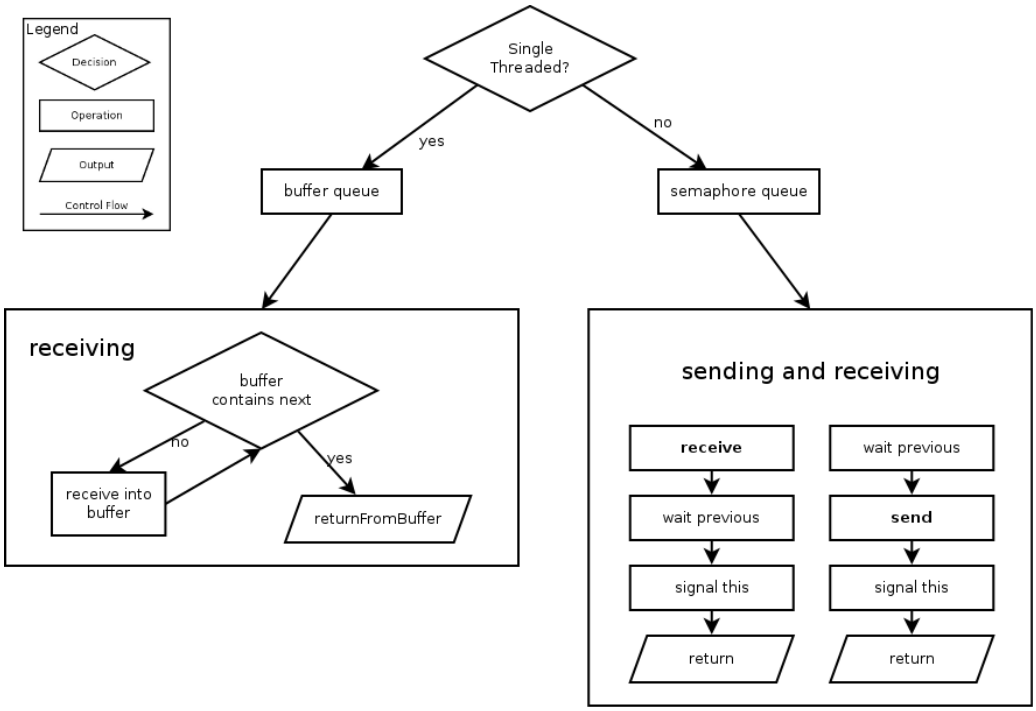

Figure 3.3: FlowChart [10] of Re-ordering Scheduler

the schedule will be consistent with earlier runs and the program state will be a proper sample.

Recording the Right Data

In order to assume that the observed program state for a particular replay is a proper sample for this replay, we have to previously record sufficient information to decide whether or not a given replay is really equivalent to an earlier one. This means we may have to run multiple replays for any given inquiry by the developer. Since we record only the partial order and communication partners there is some initial uncertainty about whether the message content is equivalent.

Before any inquiry, we collect message samples for all communication in a replay. This diminishes the aforementioned uncertainty, because, on the application layer, we can treat message contents as parameters to the remote system. This way, we can not only define a maximum variance for messages to decide call equivalence, developers can also review the message contents and decide, depending on the domain, whether the schedule is correct.

3.3.3 Online Replay

In general our replay and record mechanism works for all the different environments mentioned in subsection 2.2.1, from local developer setups to the deployed system. We consider the deployed system to be the superior basis to acquire information during replay.

A test environment already offers much better control to the developers for debugging failures, even without the application of Replay-driven Fault Navigation. Combined with our approach, we could use the test environment to replace the diagnosis script with a more "'traditional"' test, i.e a piece of executable code that resets the test environment to a known state and then replays the recorded live-system schedule within the test environment. Communication partners that are part of the deployed, but not the test system have to be mocked with prerecorded data.

The advantage of this way of using our approach is the amount of control it offers. If the setup is sufficiently close to the deployed system, the replay will inject the deployed system behavior into the test environment. Multiple replays may be executed in quick succession, as network and processing delays are small due to the close physical proximity of servers in the test environment, and the reduced load without live users that the system has to process, too. Yet, those same variables may cause the test environment to produce rare, non-deterministic failures at a different rate, or not at all. Additionally, mocking servers that are not part of the test environment introduces another uncertainty into the system, which cannot be trivially proven to have no impact on the failure. Thus, we focus on applying the replay mechanism directly to the deployed systems.

Distributed Recording

Running debugger processes across a network and controlling them from a developer machine introduces delays that prevent synchronous halt, step, and resume. We circumvent these delays by using on-demand scheduling and recording, yet, the question remains how to propagate on-demand record in a synchronized manner in the distributed system. We have not implemented such a mechanism, but different designs come to mind, for the open-world case and the closed-world case, respectively.

Our scheduler can be used to instrument some or all of the servers in the deployed system, to replay schedules in the same environment in which they were recorded. The advantage of this approach is the availability of live data and more faithful reproduction of the failure. It is non-trivial to reproduce the setup and state of a sufficiently complex distributed system in a local test environment.

It may be easier to replay in the deployed system than to re-create the deployment correctly enough to obtain the same execution behavior.

In order to recover more data from the live system during replay than we already have, the scheduler needs a mechanism specify recording on-demand. This means that, while the developer is debugging, the diagnosis script and scheduler replay the debugged schedule again and again, each time recovering one more piece of information, like for example a call stack, values of variables, and so on. Recovering only one piece at a time means that developers have to wait until the script has been re-executed and the information was recovered, but it ensures minimal disruption to the service of the deployed system. This impedes the fidelity of the approach outlined in approaches such as the Path Tools [33], which we build upon, but we show that effective debugging is still possible with this approach (cf. section 5.1).

As mentioned above, our scheduler only works for well-behaved clients in a functioning network, and in the absence of those, heuristics need to be applied to ensure continued availability of the distributed system overall. Since the scheduler cannot do more than wait for the appropriate messages to appear to shape the traffic into the right order, it depends entirely on network connectivity and deterministic clients to reproduce a schedule. These constraints are not always met, as network and server outages may make crucial parts of the schedule temporarily unavailable. In this case, the scheduler will begin to hold requests and responses on all nodes until temporary outage is over. This is not desirable, as it effectively shuts the distributed system down. We think that configurable heuristics for timeouts, dropping requests or patterns for requests that should never be rescheduled can circumvent this problem.

Closed-World Case In the closed-world case, all the servers in our distributed system use our scheduler, and we have full control over the in- and output of the system, both during record and replay. Also, no servers are added or removed between record and replay.

During recording, each server generates a universally unique identifier (UUID) for itself, from any suitable source such as the MAC address of the network card or the UUID of other hardware components such as hard disks. After the first execution, the log files from different servers are aggregated into one and their UUIDs are mapped to their recorded IP addresses. Thus, the server UUID allows us to associate a server uniquely with events in the schedule.

During replay, the desired schedule is chosen at the developer machine. The additionally desired information is added in form of queries to the appropriate schedule event. For example, at the beginning of a debugging session with a

particular schedule, we re-execute the schedule and for each event record the first few hundred bytes of data sent or received. The schedule is split into parts for each server that contributed to the original schedule and sent out to those servers, via the same mechanism used to aggregate the logs (cf. subsection 3.1.3). On each server the scheduler will then read the log and re-order network access to suit its partial schedule. Whenever an event with an additional query is encountered, the scheduler will execute the query and log the result before continuing its operations. At the end of each re-execution logs are again aggregated from all participating servers and presented in the debugger.

Open-World Case In the open-world or mixed-world case, not all participating servers include our scheduler or allow us to record at will—it may be desirable, for example for Web services, to offer only replay data, but not source code, to their developers. In the open-world case we can also allow the number servers to change between record and replay.

In this case we record all message contents between scheduled and non-scheduled servers. During the replay phase, any network event from a non-scheduled to a scheduled server is performed with the recorded data, not with the real network. The wrapped framework call for network access is not executed. Likewise, any message sent to a non-scheduled server during the record phase need not be sent again during replay.

Configuration Options

For well-behaved clients in a closed-world setup the scheduler will be able to perform its operation without problems, because given we start with the right network event, all required events will eventually be generated and the scheduler on each server can re-order events as necessary. However, in the mixed-world case, with non-scheduled servers, or in the face of unreliable network, with servers that appear or disappear between record and replay—think of Web page clients that use the deployed system while record and replay are in progress—developers have to configure the scheduler to work under these circumstances.

A scheduled server may receive events from servers that are not part of the schedule, or wait to receive events from servers that were available during record, but are not anymore during replay. We believe this problem cannot be solved in a general way, and for these cases we offer several settings with which developers can configure behavior for these cases on each server.

Timeouts Waiting for events may hold requests and responses unduly. A server that keeps no state between requests does not need to follow a schedule precisely for the re-execution to be useful. For example, in our flight reservation

service, the Web page server has no state, as all the state is in the cache and airline databases. It may be useful to release network events after a configurable amount of time, regardless of their place in the schedule.

Unscheduled Events Non-scheduled servers may send events during replay that did not occur during record. If dropping these events is an option, it may be desirable to do so in order to keep the schedule. Otherwise, bypassing the schedule for these events may also work, depending on the setup—in our example clients that access the flight search during replay can bypass the schedule, as the search has no bearing on the reservation bug.

Disruptions It may be desirable to specify a separate timeout to wait for scheduled events. In our example the cache, which is queried for searches as well as rates and reservations, should not wait too long for an event. If a scheduled event is not generated in the network in time, the schedule should be marked as disrupted, re-scheduling is aborted, and all pending events are released. In this case the re-execution fails and has to be restarted.

Message Level Developers know their deployment setup best. If the re-execution is delayed, because the scheduler is waiting for a specific event to be generated from a specific server, this information may in itself be useful to the developers, even if the event never is generated. For example, if a backup is sent to another server from our flight cache every day at noon, and this caused a failure, replaying the schedule that includes the backup upload event can only work around noon, because otherwise the event is not generated. Informing the developer about this can be enough to provide a clue about the failure, and is preferable to stopping all network communication until the next backup network event is generated.

3.4 Debugging at Different Levels of Granularity

We can record and replay network schedules in the deployed system. We use this mechanism to refining our recorded data during each replay. The kind of additionally recorded data includes network event contents, call stacks, method arguments, and return values. Developers can use this information to associate events at the network communication level to method activations at code level.

3.4.1 Connecting Network Events to Code

To debug asynchronously communicating applications developers need to move between the different levels of abstraction, from communicating servers to message sends between objects. They have to map the observed network behavior to

the original intentions expressed in the code. In our approach we support this mapping with a tool that integrates a network events with call trees.

One conventional approach to debugging distributed, communicating applications is to try and determine likely locations of defects in the code from the communication log, and the use a sequential debugger to debug each server process independently to figure out which code segment is responsible for a given unanticipated or erroneous network event. However, with Web and other applications increasingly utilizing asynchronous communication and event loops, mapping from code locations to network events and back is becoming harder. While outgoing requests can still be identified because they use only a small number of framework calls for network communication, incoming communication is increasingly handled by event callbacks. For this style of communication, a debuggers' stack view will only reach back as far as the last receive, with no possibility to look back further to inspect previous communication events.

We propose tool support to provide an integrated view of both network communication and call tree, synchronized to move freely from code to network events and back. We argued that debugging across a network of servers is similar to debugging across threads or communicating objects. It appears natural to merge the communication trees and view network communication as nested calls that originate from and return to code. A debugging tool for distributed debugging may generally support multiple, synchronized call trees, each call tree may show network events or communicating objects.

Using our approach, developers are free to move through the recorded execution both on the network level or on the code level. The distributed debugger provides two synchronized views, one for the global view on the network event log, and the other for a call tree specific to the currently debugged server. Developers may interact with the network view and, for example through clicking on an event, a re-execution is triggered. This re-execution adds a query to the schedule that logs all method activations on the server receiving the event, in case of a request, or sending the event, in case of a response. With that information, we can build an execution-specific call tree for this server, and highlight the method activation that was responsible for generating or receiving the network event. Since we have recorded the full call tree, the developer is now free to move up and down the call tree to orient himself in the code.

The debugging tool can use the information about sender and receiver for each event to show the code that corresponds to the event. That means showing the sender code for outgoing events and the receiver code for incoming events. If the source code for a particular server is available, the tool can automatically show the method responsible for each communication event in the call tree. This provides additional insight to the developers. They can then inspect method arguments

or local values, which will trigger additional re-executions and recordings as needed, to find out more about how the event data is generated or processed.

We have decided against adding support to show call trees from multiple servers side by side, because we think the network event view is more suitable to navigate from one server's call tree to another. The interfaces between the different servers are encoded in the communication data, rather than the arguments or local variables in the servers' methods, so we think that presenting multiple call trees side by side has no additional value.

Inferring Interfaces Across Languages A consequence of using our approach is the direct connection our debugger can draw between code in different languages running on different servers. Through the network events, we can infer interfaces between those different languages. Those interfaces are implicitly defined, in most REST applications, or have more explicit definitions only in another language, for example in CORBA, SOAP, or other kinds of middleware.

Without such explicit definitions so-called REST interfaces rarely follow a specific pattern for their communication. Oftentimes, the wire encoding of choice is JSON, because it is easy to use in the browser and parser implementations exist for most languages. However, JSON as a simple key-value format for data transfer offers no default interface definition language, and various services[3] offer no more than textual descriptions and examples for their REST APIs. Such descriptions are hard to verify against an implementation.

The more immediate issue with informal descriptions of APIs is, however, that outdated information is difficult to update, as this requires checking the source code and update the documentation with the informal API description. With our approach, the interface can be determined without access to the source code, by reviewing recorded message contents for valid communication. If there is a communication error somewhere in the log, and it is suspected to be due to improper usage of the API, the event's message contents need only be compared against a log of a working communication, to test this hypothesis.

Once such an error is discovered, our approach offers the direct connection from the problematic events to the source code where the events are generated and received. Thus, developers can test their hypothesis and move to the appropriate piece of code to fix a potential problem.

[3] Github API v3: http://developer.github.com/v3/
Twitter REST API: https://dev.twitter.com/docs/api
accessed July 25, 2012

4 Implementation

We evaluated our approach in a prototype for distributed applications using HTTP to communicate, implemented in Squeak/Smalltalk [21]. In this chapter, we explain how we hooked into the entry and exit points for network communication, for which we used *ContextS* [20], an implementation of context-oriented programming (COP) for Squeak/Smalltalk. We already mentioned that we built on earlier work about online analysis of runtime behavior, implemented in the *Path Tools* suite, and we have based our debugging tool and visualization on the Path Tools suite. Figure 4.1 shows how our protoypes' components interact. Applications are instrumented with ContextS to include a logging and a scheduler layer. The logging creates trace data that is used by our query engine to display the network communication patterns with the Path Tools and provide any recorded call tree data. When the developer chooses a schedule for re-execution, queries are attached to the schedule events, the prepared schedule is sent to the scheduler, and the application is re-executed. The scheduler shapes the communication during re-execution to match the schedule and records any additional information using the same mechanism as the log writer. The resulting trace data is again sent to the query engine, which passes the newly recorded information to the Path Tools for the developer to inspect.

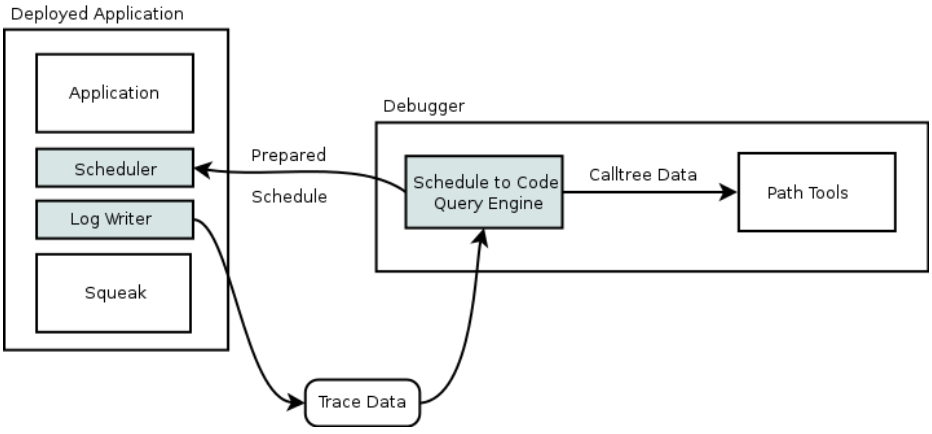

Figure 4.1: Interactions in the Prototype

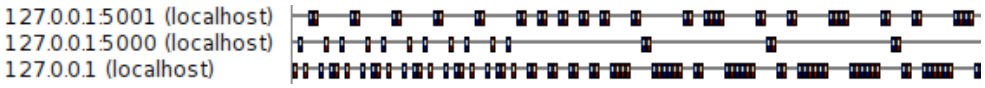

Figure 4.2: Communication Schedule Overview

4.1 Tracing Communication Patterns

One way to implement the logging of logical communication schedules is tracing the appropriate code executions in the system. For our application, we identified the methods in a Squeak image where HTTP requests are received and responses are generated. Our example application used, besides the standard library HTTP communication, the *KomHttpServer* for receiving requests.

Tracing communication at different levels of granularity was vital to verify our approach in a prototype. We chose ContextS so we could easily add various layers with different logging mechanics for experimentation. Additionally, ContextS layers are easy to manage in Monticello, the version control system (VCS) of Squeak. This allowed us to keep the tracing implementation as a separate project an thus fairly independent of the traced system.

The combined log includes all events that were recorded between two particular points in time. These points in time are given by the start and finish of our diagnosis to reproduce a failure. Re-executing such a diagnosis a number of times yields different time slices of communication. We present these logs in a diagram with events as data points and logical time on the x-axis. Because events are data points, and there may be many events in one schedule, our visualization scales all events to fit into one window. For many events, this allows developers to get a high-level overview of the activity throughout the system without swamping with detail. In Figure 4.2 we see such an overview. While we cannot make out details, we can see that the service running on *localhost* port 5001 is active more consistently than the one below it, running on port 5002. This high-level pattern communication pattern may already provide some insight as to how the system works.

Figure 4.3 shows an example of a very small communication schedule. We show the number of diagnosis runs that yielded this particular schedule (①), and how many events there are in this schedule. The color of the box shows whether there were any errors, such as failed assertions, during the diagnosis' execution by turning from green to yellow. The recorded communicating servers (②) are presented as IP address labels on the y-axis, with the associated port (if any). If that address resolves to a host name, it is added in brackets. *Lifelines* extend from the endpoints, so events on the right-hand side can be easily associated to a particular server. Each box represents an event with outgoing events (③) in

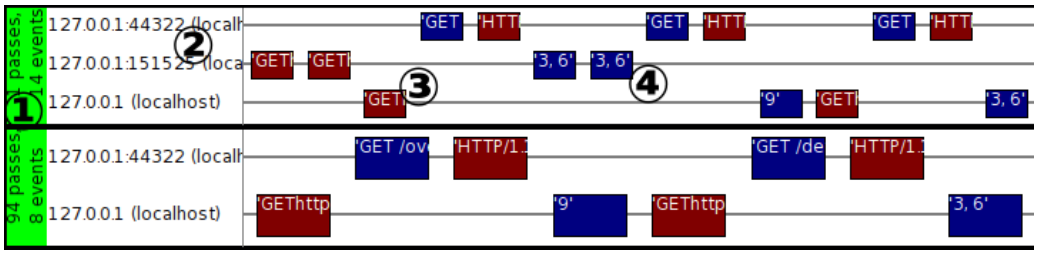

Figure 4.3: Captured Communication Logs

red and incoming events (④) in blue. The boxes in this visualization represent a point in time rather than a time span, and their width is simply as wide as possible to still show all events within the graph. Because each box shows the first few bytes of captured message data if such has been captured, it is useful to have boxes as wide as possible.

4.1.1 Distributed Context-oriented Computing

ContextS layers can be dynamically enabled and disabled to extend methods at runtime. For a distributed system, these layer activations have to be distributed across the network, so enabling a particular tracing mechanism on one system carries over to another. This can be achieved in various ways. Because our layers are implemented for HTTP requests only, we decided to propagate layers in a custom HTTP header, `X-DCOPActiveLayers`, an application specific header in accordance with RFC 2047. This header contains not the names of the currently active layers, but, as suggested in [34], an ontology describing those layers. This decouples the layer names on our particular implementation from possible other implementations and their layer names, by describing what layers do instead. Those headers will be ignored by conforming implementations that do not use our tracing package. On systems that do use our tracing, we read the list and activate the layers globally as required.

Most Web application frameworks employ the *inversion-of-control principle*, which does not work well for ContextS. In ContextS, layer activations are valid for the dynamic scope of the execution, and do not propagate upwards the stack from the point of activation. This problem with COP in the context of inversion-of-control frameworks is known and has been treated in recent work [3, 2]. We have not implemented the mechanism to deal with this problem, so our prototype is not able to activate layers across the network. Rather, the required layers are simply active all of the time.

In related work, DejaVu uses a similar mechanism on the transport layer that involves embedding and, on the remote end, stripping magic bytes in TCP packages [24] to propagate meta-information for tracing.

4.1.2 Tracer Entry Points

Tracing HTTP requests with ContextS entails finding and layering methods in frameworks and standard library that create and receive HTTP requests. In our example, using the Squeak standard library and the KomHttpServer framework, this included layering only 3 methods.

Extending the implementation to work for other Squeak Web servers or other application layer frameworks such as relational database adapters is trivially possible. The actually captured data only includes the local logical timestamp, as well as the sender and receiver of the message.

4.1.3 Trace Format

To actually implement our approach for different runtimes and debug across languages, an interchangeable trace format is necessary. We have not implemented such a format. Our prototype is Squeak-based, so the interchange format consists of a Squeak store string, which can be recompiled into a collection of trace events. Each trace event, here, is an object with a logical timestamp, receiver and sender IP and port.

An interchangeable trace format is trivial to implement, however. To exchange logs, we only need a way to transfer collections of event 3-tuples that include strings and integers which almost any data-interchange format should be able to. JSON seems to be a natural choice, both because it is widely used and parsers are readily available for a variety of languages, and because it is the most common format in Web development.

4.2 Communication Diffing

As per our design (cf. section 3.2), we regard each communication schedule as a sequence diagram with only asynchronous message sends. To help developers find differences in the schedule, we want to automatically differentiate traces to show areas of high and low variance in the schedule. For N servers, at any one logical time, there are at most N network events. Comparing different communication schedules can thus be done by comparing each logical time in order, similar to how a patch algorithm compares files at different versions.

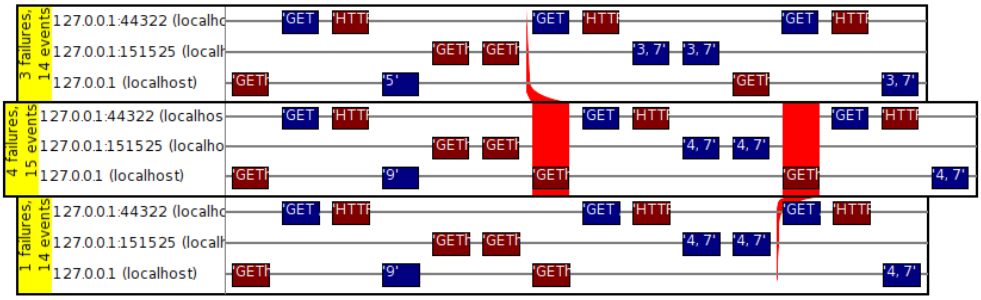

Figure 4.4: A Simple Differential Between 2 Schedules

4.2.1 Difference Format and Visualization

The choice of algorithm for automatic differentiation can have a huge impact on the usefulness of the result. There are various algorithms to differentiate tree structures [9]. We treat our communication schedules as call trees with only asynchronous communication. That means they can be transformed into a linear form, which makes them suited for the linear differential algorithms used for file comparison. Of those, we chose the longest common sub-sequence (LCS) algorithm to create differential schedules.

We can assume that schedules will vary in some events or event-sequences, but will be equivalent for some length around those variances. LCS dynamically finds the longest matching sub-sequence between two pieces of data. For comparing two schedules, we choose one as the *base* and the other as the change. In Figure 4.4 we show such a differential schedule between the base at the top and the comparison schedule at the bottom.

The differences between the schedules are marked in red. If we compare the overall number of events, we can see that both the base schedule at the stop and the comparison schedule at the bottom have the same number of events. They are equivalent up to the first red marker, where the comparison schedule sends out a `GET` request that does not occur in the base schedule. If we follow the differential, however, we can see that the schedules are equivalent again for four more events until the second red marker. At that point, a `GET` request sent in the base schedule does not occur in the comparison schedule. Afterward, both are equivalent up to the end.

4.2.2 Correlating Variances and Failures

Not all variances will usually contribute to the failure, yet, for large numbers of runs we may have more variances than common sub-sequences. Simple red/white differentials are not very helpful in this case, as they do not signifi-

cantly reduce the search space to aid developers in their search for the problem. Due to this, we weigh variances according to a number of rules, which have been in described in subsection 3.2.2.

The heuristic probability distribution is mapped onto hue values from zero to sixty degrees, which corresponds to full red and full yellow, respectively. For variances that the heuristic recognizes to have no impact on the failure, we assign the color gray.

To review variances between passing runs, we assign hue values depending on the amount of variance on each particular network event. The function *hue*, below, shows how we map the each event onto a range from 0 to 1. We then map this range onto a range from 0 to 60, corresponding to the colors from red to yellow in the HSL color space. We assign alpha values from 0 to 1 accordingly, which means that events with a hue closer to 60 are hardly visible.

$$
\begin{aligned}
T &= \text{set of trace event sets} \\
N(event) &= \{T_i \in T | event \in T_i\} \\
P(event) &= \{T_i \in N(event) | T_i \text{ is pass}\} \\
hue(event) &= \begin{cases} \left(1 - \frac{|N(event)|}{|T|}\right)^2 & \text{if } N(event) = P(event) \\ \sqrt[12]{\left|0.5 - \frac{|P(event)|}{|N(event)|}\right|} & \text{else} \end{cases}
\end{aligned}
$$

The result of this mapping, shown in Figure 4.5, is that high-variance events from traces that are all passes or all failures show more red, which in return means that low-variance events have almost no highlighting. In sets of traces that include both passes and failures, the second branch of the *hue* function is used to determine a value that highlights events that occur not in all passes—thus are not likely to be the cause for passing—but also not only in failures.

4.3 Reordering Access to Network Resources

In our prototype we support re-scheduling network access in the live system. Parallel network events inherently cause races, the network transport may transmit messages at different speeds and the operating system may forward messages to user space applications as it sees fit. To reorder access to network resources, the network events have to arrive at the application in order, so network events have to be re-ordered in a layer between the application and the hardware.

We decided to re-order events in the language, using the same mechanism as for tracing. We use ContextS to define a scheduler layer between applications and

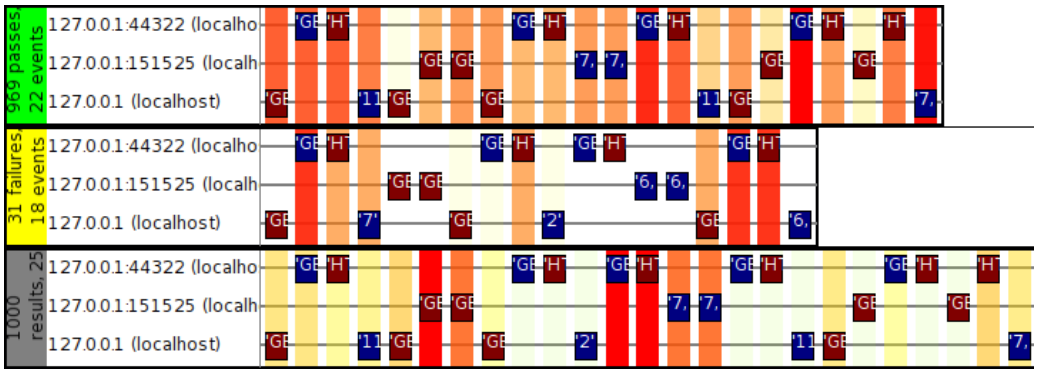

Figure 4.5: Hue Mapping For Events

the KomHttpServer framework and the Squeak standard library (cf. Figure 4.1). This scheduler layer is only active if a schedule was requested.

4.3.1 Constraints on the Networked Systems

Our design is supposed to work for local systems on the developer machine, in test environments and the live system. However, because we only record communication partners and the partial ordering of events, in practice, there are various constraints on those systems. First, hey have to be deterministic in their communication, which means if we send them messages in the right order, they return valid responses in right order. This means have to be able to control the order in which messages are received. However, with systems that employ load balancing, multi-threading and event loops which are designed to handle multiple requests in parallel we cannot be sure that for multiple requests the responses are generated in order. This means we have to control the send order, too.

4.3.2 Entry Points

According to the design, there are two types of entry points which we need to control to enable re-scheduling, both for sending and receiving. We only deal with HTTP requests and, because we use the same context-oriented mechanism as for the tracer, we only had to create a layer to wrap the same three methods as for the tracer. Our scheduler layer is always active and the wrapper methods check whether a schedule was requested. This check simply uses a globally available singleton object and checks whether it contains a valid schedule. If not, the scheduler methods do nothing and simply proceed.

4.3.3 Scheduling Execution

Our implementation follows the design shown in Figure 3.3. The framework we used to implement our example application uses threads to handle multiple connections, so for simplicity, we have only implemented the threading scheduler using semaphore queues.

To schedule an execution, the trace that is used as basis for the execution is copied and transformed into a list of schedulable events. All of these events contain the identifiers for the original communication partners, the id of the event and a semaphore in the *waiting* state. This list is then attached to the globally accessible singleton object and the first semaphore is signaled. Finally, the diagnosis script is started to provide the required events.

Receive The KomHttpServer uses a blocking *accept* call on a socket object to wait for incoming requests, which the virtual machine (VM) translates into the *accept* system call. When that call returns, the operating system socket has data available. The KomHttpServer framework then spawns a new thread that creates an *HttpRequest* object from the received data and passes this into the application.

Before it passes from framework to application code it has to go through our scheduler. The scheduler uses an atomic operation to find the appropriate event in the schedule for this request, attach the request object onto it, and return the event object from the schedule. It then calls *wait* on the event object's semaphore. If the semaphore was already signaled this call returns directly, otherwise this thread blocks until the semaphore is signaled.

When the *wait* call to the semaphore eventually returns, the semaphore for the following event is signaled and the request object is passed back into the framework, from which it passes on into application level code.

Send Sending of data works similarly. The application manufactures an *HttpResponse* object and returns it to the framework. The framework the calls the *writeOn:* method with the socket object to serialize the response onto the socket stream. We wrap this method and use the atomic find, set, and return operation on the schedule to find the scheduled event corresponding to this response, and then wait on the semaphore. When semaphore is signaled, we proceed to the base method which writes the response to the network.

The fact that we have only implemented the semaphore list for our scheduler means that means it is currently limited by the maximum amount of threads available to the framework for handling multiple connections. Because each thread blocks if the event cannot be released directly there may be situations in which so many threads are blocked that the framework cannot create any

more threads for incoming requests, at which point the server will deadlock. We circumvent this problem in our implementation by having the semaphore wait time out after a configurable amount of time, after which point the re-scheduling deviates from the desired schedule.

4.3.4 Probable Replay

A goal for a scheduler following our design is to enable consistent replay. Our design only works for well-behaved clients and stable networks, constraints that cannot always be guaranteed in a real environment. However, these constraints may often be met, even if not all the time. A practical re-scheduler may not always work, but it raises the probability that a specific schedule will be reproduced, and it will reproduce the schedule if the circumstances permit. If the network is sufficiently disturbed, either because of misbehaving clients or connection problems, the scheduler deals with it by deviating from the desired schedule.

As mentioned above, our implementation uses timeouts to make sure that the distributed system does not lock up, but timeouts include the potential for releasing events too early and thus deviate from the desired schedule, causing a re-execution inconsistent with the original trace. Our implementation also continually checks for disturbances defined in section 3.3.3—unscheduled events, schedule disruptions by misbehaving clients—and we have implemented configuration settings that determine how we deal with them. These settings currently allow developers to choose to drop unscheduled events, or pass them through without waiting for a semaphore. Our implementation additionally uses simple printf logging to communicate to the developers how it is reshaping the network schedule, i.e. which events have been encountered, which are being waited for, and which disturbances were encountered and how were they dealt with.

4.4 Connecting Network and Object Communication

Adequate tools are essential for efficient debugging. Previous approaches at efficient replay for the most part focus on recording and replaying correctly and efficiently, but use existing debuggers like *GDB* for interactions with the program state, and separate tools to review the captured network data. Other approaches, like *Causeway*, focus on the tools.

We think that the debugging tool should be integrated, allowing the user to inspect both object and network communication in one view, as they are in Causeway. However, efficient replay is required for those tools to be practical, so a recording and replay mechanism like our scheduler has to be integrated into the tool.

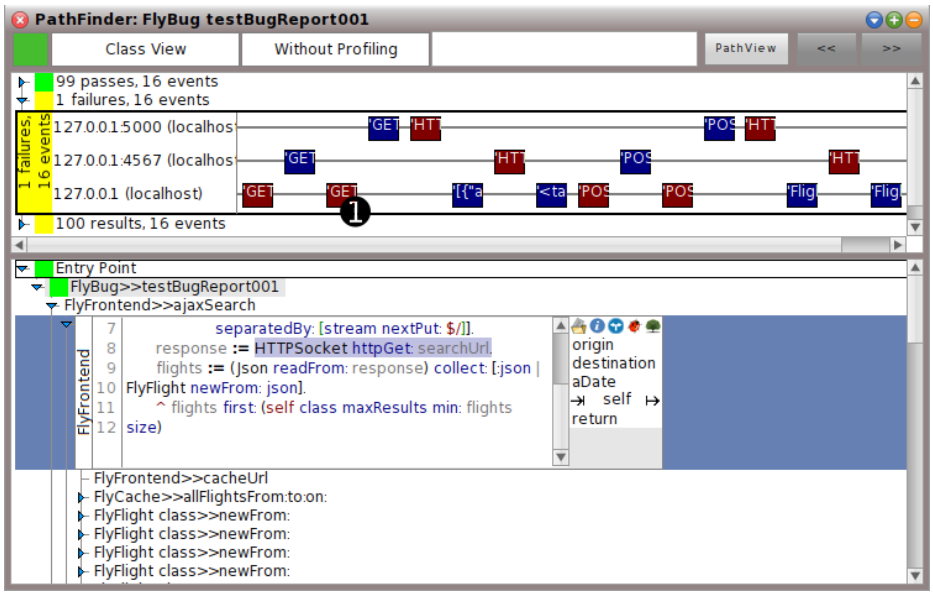

Figure 4.6: The Path Finder Network Extension

Our prototype extends the call tree view, called *Path Finder*, which is part of the Path Tools suite. Our tool, inspired by Causeway, shows network communication and the call tree for one server in a single window. These views should be synchronized, so if a user opens a method, the closest network event should highlighted. Inversely, clicking on a network event should open the call tree on the method that is closest to the network event. If the network event was sent or received on the server to which the call tree belongs, it should show the method responsible for the event.

For this prototype, we have only implemented moving from the network to the code view, not vice-versa. If a user clicks on a network event, the scheduler is instructed to record the stack at the time of the network event and the test case is re-executed. We then search for a method on the stack that is part of our call tree, and open that method for inspection. In Figure 4.6 we show the result of clicking on the third network event ①. In the lower part of the window, the call tree is expanded to the method which sent the event.

A full implementation should support full synchronization between the two views. We have also not implemented additional queries of the program state. A fully implemented tool should support queries of single variable values, method arguments and other program state in the scheduler, to allow further inspection of the program state.

5 Evaluation

We have presented the example of a flight search and booking system in section 2.1. For this system, we presented a possible bug report. In the report, a client searches for connections and the service returns a list of flights on the given date, with available seats and pricing. When the client clicks on the link to make a reservation, the reservation goes through, but possibly at a higher price.

Our flight search interface uses AJAX requests to keep prices and available seats up to date while the client is viewing Web site. However, the inherent problem in this system is the possibility for data races between price updates, the responses to the AJAX requests of the Web site, and the click on the link to reserve a seat on a flight. These data races cannot be eliminated in a distributed system, and thus have to be dealt with. In order for developers to be able to deal with them, however, they have to understand that these races may occur.

Traditionally, to prevent these kinds of failures, the developers would reason about possible event orders and their effects on the global state of the distributed system. That means the developer may use network diagrams and the documentation of the various services—in this case, the airlines' flight information and booking systems—to think of scenarios that may deviate enough from the default (the price does not change in the, on average, short time between search and reservation) that data races with undesired results may happen.

In this example, some amount thinking will lead to the recognition of the data races that incur the discrepancy between the price for the search result and the reservation price. Some more thinking will lead to the conclusion that a transactional lock, in effect a reservation with an option to back out without cost, may not be possible with at least some of the airlines. A possible solution to this problem may be to inform the client about an intermediate price change before going through with the reservation.

Given this solution, a developer with knowledge about the implemented system has to map this high-level description of a data race onto concrete code. In this example, a reservation request has to check the current price against the price returned for the search result. The code for the reservation is shown in Listing 5.1. The system receives a POST request to reserve a seat on a particular flight. It will forward this reservation request to the airline gateway on behalf of the client. If the reservation is not entirely successful, indicated by a status

code other than 200, the failure code is forwarded to the Web client, together with the contents of the airline gateway's response. To fix the above defect, this method has to be changed to accept not only the flight id for reservation, but also the desired price. The request can then be forwarded and if the price is not the expected, an additional confirmation Web page can be returned instead of the content of a successful reservation.

```
reserveSeat: flightId
  <post: '/flights/%/reserve/?'>
  | response code data |
  data ← 'id=', flightId.
  response ← (HTTPSocket httpRequestHandler)
              httpRequest: 'POST'
              url: self airlinesUrl, '/reserve'
              headers: ('Accept: */*',
                        String crlf,
                        'Content-Type: application/x-www-form-urlencoded',
                        String crlf,
                        'Content-Length: ', data size,
                        String crlf)
              content: data
              response: [:rr | "Extract the response code"
                         code ← ((rr copyUpTo: String cr) findTokens: ' ')
                                  second asNumber].
  code ~= 200 ifTrue: [self status: code].
  ↑ response contents
```

Listing 5.1: Flight Reservation Request Entry Point

Using our prototype, we argue that these steps can be simplified. The flight system can be written with the most common case in mind, namely that the price will usually not change between search and reservation. Deployed in a test system, the continuous tracing can be enabled during load tests. As such tests will execute many searches and reservations in parallel, the data races leading to the above bug will certainly be among the recorded network schedules. In contrast to a diagnosis script written after a bug report, such network schedules may not be marked as failing, because the load tests will not include assertions for all undesired behavior, but only such behavior as the developers have thought of. That is not a problem, however, because just differencing all schedules recorded during load testing will yield a diff that can be used to reason about possible data races. Thus, reviewing the schedule enables developers to review the actual communication data generated in their system, rather than having to reason about possible communication patterns.

Differencing all available communication data may still yield a lot of noise. This can be improved if the load tests are enriched with assertions of expected behavior, such as status codes, Web page contents like prices, available seats, and such, as well as assertions about request order. Using these assertions, our

prototype can help developers divide the observed variances in communication schedules into critical and non-critical variances.

Given any description of an undesired communication pattern, either through reasoning or using our trace differencing tool, we also help developers implement solutions to the various problems. In the traditional case, after deciding upon a fix for the problem—presenting an additional confirmation page if the price changed, for example—developers need knowledge about the system and the employed Web framework to know which code represents the entry point to implement the fix. Using our prototype, this can be simplified. Given network traces, developers can just click the network event that represents the reservation request, and our prototype will show the corresponding code in the receiving system. This will lead them directly to the code in Listing 5.1, even without prior knowledge about the system.

In the following sections we evaluate aspects of our prototype implementation corresponding to the goals we set ourselves, namely:

1. We need a reasonably lightweight tracing mechanism
2. Analyzing schedules should enable us to provide hints to developers about where to start debugging
3. Shaping network access in the running system has to be stable
4. Server events are connected to code locations in our tooling

5.1 Tracing Overhead

Our prototype uses ContextS to implement the tracing, which itself has a very high overhead. To get an idea how much overhead ContextS imposes just for method dispatch, we have run a send heavy benchmark, calculating the Fibonacci number of 26 recursively, both without and with layer on a recent build of the Squeak CogVM. The CogVM includes a JIT compiler with polymorphic send-site caching, which, after some warm-up, makes its method dispatch performance comparable to other languages with a just-in-time (JIT) compiler, such as Java or C#. For our benchmark, we ran the unchanged recursive Fibonacci implementation both unchanged and with a ContextS layer active, as well as the layered implementation, again both with layer on and off. We ran each benchmark 10 times for 5 seconds each. We recorded how many times per second the Fibonacci of 26 was calculated and averaged it over the ten runs. In Table 5.1 we show the results of this benchmark, rounded to 3 significant digits. From these numbers we can gather that just activating ContextS executing through non-layered methods introduces a large slowdown with a factor of 62.4. Just having layered method

without any layer being active produces an insignificant slowdown of only 0.03. Note also that execution times for layered methods vary more than non-layered methods.

Benchmark		Avg Executions/s	σ in percent of avg
No layer active	Non-layered	52.2	0.847
	Layered	50.9	2.08
1 Layer active	Non-layered	0.836	0.335
	Layered	0.639	6.18

Table 5.1: ContextS Benchmark

This drastic slowdown seems to preclude any intention of using ContextS for low-overhead tracing. Note, however, that a recursive Fibonacci is a very send-heavy benchmark, which escalates the overheads of the ContextS implementation. For tracing, we need only layer a select few methods (cf. subsection 4.1.2). Additionally, we argue that most of the time in a Web application is spent waiting for IO, and not the CPU. Our second benchmark thus consists of running a diagnosis script for the above reporting example both with and without tracing, also 10 times for 5 seconds each. The results of this can be seen in Table 5.2. We think it is clear that the tracing overhead is minimal, with the tracing inducing a slowdown of only 0.01. The variance is comparable, as could be expected because the entry points were layered for both benchmarks. These performance results show that our prototype fulfills the requirement for low-overhead tracing, because the vast amount of time is not spent in method dispatch, but rather waiting for the network. These benchmarks were run on a local machine connecting through the loopback device, so in a distributed setup the overhead may be even more insignificant.

Benchmark	Avg Executions/s	σ in percent of avg
Normal execution	8.51	3.64
Tracing	8.39	1.36

Table 5.2: Tracing Benchmark

These results may be interesting for implementations of our approach on other languages. ContextS is one of the slowest available implementations of context-oriented programming [4]. Given these results, a context-oriented implementation for other languages should equally have a very small practical overhead.

5.2 Case-Study: Reporting System

To evaluate our approach with another system besides the flight search and booking example, we have adapted a system that is deployed in industry. The original system is written in Ruby on Rails and deployed in the intranet of a large company, with about a hundred users. The system implements a reporting Web service, which its users query to get current data about other systems within the company. A user will generally have a few prepared reports that provide a high-level overview various systems for different countries, industries, and company sectors. If any of the overview reports are not within expectations, the user will drill down into the reports to see more detailed listings of the available data.

The technological challenge in this system are the large amounts of data which need to be processed as fast as possible to provide the most recent data to its users. The data is aggregated over various servers throughout the world, and preprocessed along the way before being imported into the reporting system. The whole process of updating and reporting takes about 30 minutes. The system overview is presented in Figure 5.1. The client communicates with the reporting server through a Web site ①, where reporting data is represented in HTML tables generated by the reporting. The reporting data is transmitted in chunks, so the summary numbers are transferred first, and the detailed numbers for drill down are transferred on-demand if the user decides to drill down. The reporting server obtains the reporting data from one of the databases, called *Yin* and *Yang* ②. One of the databases will always be the *active* database from which queries are served, and the other is used by the importer to provide the most recent data ③. This is to ensure that queries are always done on complete data sets. Once an import is done, a switchover command is sent to the reporting server, which exchanges the roles of the databases.

This system was deployed and some time later, the developers received a bug report very similar to our example report. In this report, one of the users noted that very rarely the sums in the overview will not match the numbers returned when drilling down into the data. This rare bug was triggered if, between requesting an overview and a drill down, the data import finished and the second query was run on different data. The eventual solution in the real system consisted of a simple message informing users about the updated data.

We have reimplemented the system in Smalltalk, keeping the setup of servers and their responsibilities and communication the same as in the original, for evaluation with our prototype.

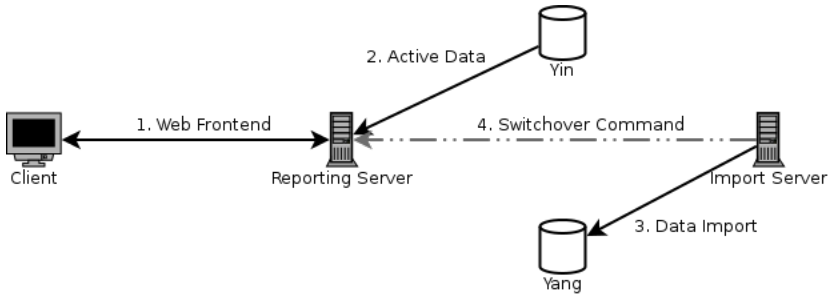

Figure 5.1: Reporting System

User Trial Our user trial was done with a Smalltalk programmer on the aforementioned reporting system bug. The different servers contributing to the system were run on one system for simplicity, and the bug report had already been converted into a diagnosis script executable through our test runner (cf. Listing 5.2).

```
testBugReport001
    | details overview totalSum detailsSum |
    totalSum ← detailsSum ← nil.
    overview ← HTTPSocket httpGet: 'http://localhost:',
                DIODataWarehouse instance portNumber, '/overview/'.
    totalSum ← overview contents asInteger.
    (Delay forMilliseconds: 100) wait. "Manager needs some time to click"
    details ← HTTPSocket httpGet: 'http://localhost:',
              DIODataWarehouse instance portNumber, '/details/'.
    detailsSum ← ((details contents findTokens: ', ') collect: #asInteger) sum.
    self assert: detailsSum = totalSum.
```

Listing 5.2: Diagnosis Script

The participant did not know the code for the system beforehand, but we have explained to the purpose and setup before the trial. After explaining the bug report and that it was transformed into a diagnosis script, the participant started his debugging session.

The participant knew the Path Tools suite and was able to use our modified Path Map to run 100 executions of the diagnosis script. Of those only 2 failed, both with the same schedule. The participant started examining the passes and the failure. He quickly realized that a minimal passing schedule only includes 3 lifelines. He was able to identify those lifelines as the diagnosis script, the reporting server, and a data store. He did this by clicking events on each and examining the source code associated with the events.

He realized also that a failure included 2 more servers and recognized those as another data store and the data importer by jumping into the code and comparing with the system overview.

The connection from network events to code were used also to determine the request-response relationships of events. At this point the participant remarked that being able to add those connections manually would be useful.

The participant then explained that he did not understand the significance of the colors in the diff traces. In the overall difference most events were highlighted with a high alpha, but two events were close to full red and had no transparency. We explained that those colors show variance in the traces and the stronger highlights should provide good entry points for debugging the problem. The participants clicked one of the highlights and ended up in the method that returns the drill down details for the report overview. He realized that this method returns data from the data store to the client. He realized that the problem is in concurrent modification of that data and checked that assumption by establishing request-response relationships in the failure schedule. He noticed that the drill-down and the data import switchover overlap. At this point we finished the trial.

Using our tool, the participant was able to find out the communication schedules for a system unknown to him. He was also able to assign endpoint addresses to servers. Multiple open problems were identified:

1. Our visualization of the differences between schedules is too hard to understand. Visualization was not the focus of this work, however.
2. While we cannot automatically establish request-response relationships for any asynchronous communication, developers should be able to connect events manually when they find out how they are connected. For larger schedules it is impractical to remember all connections.
3. To understand the concurrent modification problem, we think data flow analysis may be useful to identify network events which contribute to a particular data point. In our trial the participant may have found out more quickly how the report data is modified once he arrived at the method that returns the data.

Overall our approach seemed to be useful, but more tool features and better visualizations are needed so developers can use the approach on their own. As it stands now, a tool expert has to be present to explain the *intended* way the tools should be used.

5.3 Scheduling Accuracy

Our approach hinges on the accuracy of our IO scheduler to shape traffic in the live system to match a previously recorded trace.

The scheduler can only uses events generated in the network, because it is not recording enough data to replay events from memory. In our examples the servers were running the full time. Some created events on an independent schedule, others reacted only to requests. The initial requests at the beginning of each trace were given by the bug report or user interaction with the Web frontend. However, this is a serious limitation in systems were not enough events are generated to re-enact a given schedule, and some claim that this may be the more common case [17]. We are not aware of any studies to support one view or the other, but if indeed servers in a distributed system become unavailable often enough—due to net splits, maintenance, or any other reason—it would render our re-scheduling approach ineffective.

A possible solution in that case may be to additionally record message contents as well as communication partners. While this may come at a larger overhead in both space and execution time, it would enable replay of single servers independent of the other systems, if, during replay, only the recorded message contents are used instead of actual live data.

For our scheduler we have implemented a number of settings that can be tweaked to make successful reproduction of a schedule more probable. First, unexpected requests may be dropped during replay. Depending on the system, this may make a re-execution more probable, at the cost of dropping possibly valid requests in the live system. Another option for those requests is to hold them until the schedule is finished and release them later, which will increase the latency of the system. Another setting controls the timeouts for waiting for an expected event to occur. These timeouts may potentially be too long or too short for a given schedule, so schedules may sometimes fail to reproduce if they trigger timeouts. Tweaking those timeouts was sometimes necessary in our trials for successful replay.

Given those settings, a hybrid approach between our approach and fully isolated replay from message contents may be preferable. If we record message contents, they could be used as a better fallback mechanism in these cases where an expected event does not occur within a specified time frame or where unscheduled requests require responses that are not part of the schedule. This may not only increase the robustness of our re-scheduling approach, but also further minimize disturbance of the live system, because unexpected requests at least receive a valid (if outdated) response, instead of a manufactured value that will, in most cases, not be a valid response.

5.4 Events to Code Mapping

Our prototype offers an overview of the global network events, as well as a call tree. Upon clicking on a network event, the scheduler re-executes the communication an logs the first 100 stack entries for each event. We use the Path Tools mechanism to find methods belonging to a certain project and search from the top of stack until we find one of those methods or give up.

In practice this works well only for synchronous requests or if responses are handled in closures connected to the method which sent out the request. In the KomHttpServer framework, however, request and response objects are created in the framework and the application handlers are called at a later point. So at the time the scheduler encounters an event, the application method is not on the stack and can thus not be automatically connected to the event.

To work around this limitation, we need a way to wrap all application methods in one-shot wrappers to record their activation. Those activations can then be associated with either the last incoming request or the next sent response.

6 Related Work

Because our approach combines work in various areas of distributed systems debugging, from continuous record and replay, over replay-driven fault navigation, debugging at different levels of abstraction and full-on distributed debugging, we have sub-divided our treaty of related work in this chapter into sections for each of those domains.

6.1 Record and Replay

The need for consistent record and replay to debug certain kinds of distributed applications is widely accepted. Approaches vary mainly in how much they record, and subsequently how much of the system they can simulate to eliminate behavioral disturbance.

Replay with Mocks Mocks are program entities designed to simulate the behavior of a real entity. In system replay, mocks may be used to stand in for network nodes. If enough information about the data sent by a particular node has been recorded, there is no need to re-execute that node during replay. Assuming the node works deterministically, a mock can return the appropriate responses.

This approach is advantageous for nodes that are not under debugger control. On those machines, package races on the network cannot be rescheduled upon receive and behavior may still vary during replay. Mocks avoid that problem, at the cost of greater recording overhead at least in space, if not in time, for copying message contents.

We have opted not to use mocks in this work, but other approaches use mocks for some or all the nodes during a replay.

Deterministic Replay on the JVM with DejaVu *DejaVu* [24] is a record and replay framework for distributed Java applications. It uses modified Java virtual machines (JVMs) to execute java applications and record their thread schedules as well as network access. Thread and network schedules use implementations of Lamport Clocks to establish event order.

The level of abstraction for network events in DejaVu is lower than in this work, recording TCP instead of HTTP events. While this enables replay just as well, we argue that it is harder for a programmer to review TCP traffic and connect it to the program domain than it is with HTTP events.

Contrary to this work, DejaVu supports both open and mixed world record and replay. we note in section 3.3.3 how the same may be accomplished in my implementation.

Tracing and Simulation with iDNA *iDNA* [8] is part of Microsoft-developed framework for instruction-level tracing and replay of applications in user-mode. The solution includes running instrumented programs atop a runtime called *Nirwana*, which provides hooks for iDNA to register points of interest for tracing. The Nirwana runtime uses dynamic binary translation to break a monitored process' instructions into smaller instructions that are then instrumented and executed on the host. The iDNA tracing component records traces via callbacks from the Nirwana framework and, for replay, reads trace records and feeds Nirwana desired instruction pointer and register values for replay. The overhead is about one order of magnitude, which makes it feasible to run IO-bound applications in tracing mode all the time, which is also the scenario envisioned by the authors.

Due to its low level of recording, iDNA supports multi-threaded applications, as well as applications that generate executable code at runtime, such as JIT-compiled languages, but does not target networked applications specifically. Similar to the present work, iDNA is aimed at tracing continually on the live system, but the focus is on single process applications rather than communicating processes or servers.

Reproducible Failures with ReCrash *ReCrash* [5] is a framework to observe program execution in Java and generate test-cases that reproduce observed errors. Similar to this work, it can be applied to live systems. ReCrash does not try to preserve enough information to replay an observed failure from the beginning, but rather just keep enough information around to provide an method entry point with arguments that trigger the crash.

To do this, ReCrash keeps a n-deep *shadow stack* during execution with copies of the receivers and arguments. If a crash occurs the stack is saved. In a second step, the shadow stack is searched for the first call that reliably reproduces the exception.

The ability to reproduce crashes using this technique is limited by the size of the shadow stack, as well as the copy depth of receiver and arguments at

each stage. Deeper stack and copying might be more able to reproduce a crash, but imposes a larger overhead. The authors offer a way out by only enabling monitoring after a crash occurred for the first time, and only for methods on the stack at that point. Since in this mode *no* information has been saved about the execution, the crash has to occur again, which might take a very long time for rare exceptions.

For the class of problems that are related to causes that are not on the same stack, like the concurrent update problem presented in this thesis, ReCrash has only limited value.

Output-deterministic Replay with liblog The *liblog* [17] tool for distributed C/C++ applications uses a `libc` wrapper library to intercept system calls and record events such as network communication, file access, and IPC.

The goal, as in this work, is to record sufficient information in the live-system to replay execution at will for later debugging. The authors decided against *in-situ* replay, however, and use a central console to download all logs and program checkpoints to re-instantiate the program locally. Additionally, they record enough information to replay each process independently of the others, which imposes a higher tracing overhead.

Like DejaVu, and unlike this work, liblog records events at a very low level. Most programs written for the Web, for example, do not use socket programming directly, so the programmer will still have to match the recorded network communication to the level of abstraction in his code.

6.2 Replay-driven Fault Navigation

Test-driven Fault Navigation with Path Tools The *Path Tools* [32] are a research project in Squeak/Smalltalk and the basis for our prototype. Path Tools uses re-executable entry points, mostly tests, to exercise code paths in a project for analysis. These entry points need to execute deterministically in order for the Path Tools' analysis to work correctly.

In our work we have extended Path Tools and use it for re-execution and analysis of distributed applications. Because Path Tools currently need deterministic execution paths to work, this work offers an approach to make network communication deterministic.

6.3 Debugging at Different Levels of Abstraction

Network Analysis and Debugging with Firebug *Firebug*[1] is an open-source debugger and Web application analysis tool for *Firefox*[2]. It offers a JavaScript debugger with breakpoints and a network pane to review XMLHttpRequest (XHR) requests made from the JavaScript code of the web-page. Firebug includes the ability to log network communication and set breakpoints on incoming and outgoing connections, which enables developers to move from network to source code, but not the other way around.

Firebug, like traditional debuggers, works synchronously on the code and network communication of the current Web pages' JavaScript process, and offers no integration with other services or debugger instances. Also, while it is possible to break on network events and move to the source code callback for a given event, there is no further integration between the two [7].

Message-oriented Debugging with Causeway *Causeway*[37] is a message-oriented debugger that tries to unify the network request/response view with the message send/receive view of the software. The authors' approach is to trace program execution as well as network communication partners and message contents, and present the captured information in a post-mortem debugger. This debugger synchronizes the network and the stack view so moving in one will update the other (cf. Figure 6.1), a behavior which has been the main inspiration for how we show the two levels abstraction in our prototype implementation.

In difference to this work, Causeway is strictly post-mortem. That means it has to trace all required information, all the time, imposing a very high overhead, a problem the authors mention themselves.

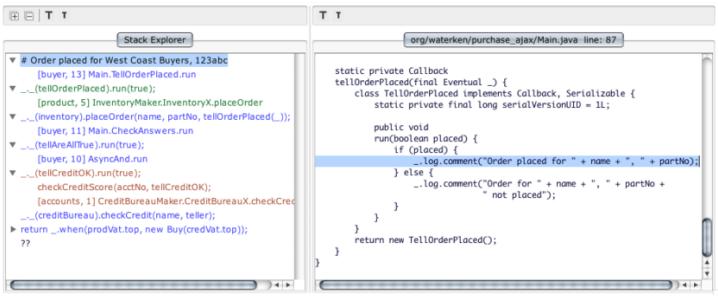

Figure 6.1: Causeway Message Debugger

[1] Mozilla Firebug: http://getfirebug.com/ accessed June 20, 2012

[2] Mozilla Firefox: http://www.mozilla.com/ accessed June 20, 2012

6.4 Distributed Debugging

Most debuggers work one thread at a time, and only for one process, and offer the programmer a way to switch threads while debugging. Most debuggers allow developers to stop all threads at once, however, this disturb the execution order and lead to heisenbugs, since uniformly starting and stopping multiple threads is not possible, even within a single process. This equally applies to processes that are distributed across a network. However, some runtimes, such as the JVM or the Common Language Runtime (CLR), but also distributed systems middleware, such as CORBA, still allow developers to stop and inspect processes across a network.

Virtual Machine Interfaces on JVM and CLR Both the JVM and the CLR offer debugging facilities to connect symbolic debuggers to remote processes. These facilities, in case of the JVM using a custom Java Virtual Machine Tools Interface (JVMTI)[3] and in case of the CLR built on the Component Object Model (COM) standard [38], abstract from the network to offer the same debugging experience across a network as for local processes. Debugging multiple processes from the same VM is possible with these approaches. Additionally, both of these VMs are built to support different languages, so debugging across languages is possible, too. However, we know of no debugging tool for either VM that supports different languages.

Compared to this work, these debugging interfaces have two shortcomings: first, to inspect a remote process that process has to be stopped by the debugger. It is undesirable for a deployed system to stop providing its service because a debugger is connected to it. Second, these debugging facilities only provide a view on the program, and do not track network communication between processes, so it is still difficult to get a global view of the distributed system.

Debugging Distributed Objects with CORBA and SOAP As a logical extension of object oriented programming into the realm of distributed systems, middleware layers, such as CORBA [41] and SOAP [40], have evolved. They allow developers of distributed systems to specify object interfaces for publishing and accessing distributed services in an interchangeable format. Developers can program against these interfaces as if the distributed system were running in a single process, while the middleware takes care of communication. However, much of the development time complexity hiding provided by middleware is

[3] JVM Tool Interface Version 1.0
http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html
accessed July 31, 2012

lost when debugging, because developers cannot step into message sends that are converted to network communication by the middleware. Recently, effort has been expended to enable symbolic debuggers to overcome this restriction and step across servers [27, 22] when debugging on top of such middleware.

Such debuggers for middleware systems, like the middleware systems themselves, try to abstract from the network. Approaches at stepping across servers are extensions to symbolic debuggers that are meant to allow developers to inspect objects in distributed system as if they were locally available. In contrast to this work, these debuggers target systems that are fully under developer control and can be stopped at will for inspection. Like traditional symbolic debuggers, they ignore Heisenbugs caused by the debugging process and are not well suited to debug non-deterministic failures.

7 Conclusion

We have presented an approach—Replay-driven Fault Navigation—to debug distributed applications and a prototype for distributed Web applications communicating via HTTP requests. Our approach covers continuous logging of network events, log analysis, communication replay, and connecting events to code locations.

Lightweight, continuous logging in the deployed system provides insight into failures that occur infrequently or only in deployment. At the same time the logger disturbs the execution only minimally. We achieve this first, by logging all the time, and second, by logging only information strictly necessary for replay. This way, the execution behavior during logging becomes the normal behavior. The minimally necessary information is the partial ordering of events and the communication partners. We have shown that the overhead of this approach is minimal and justifiable for deployed systems.

Our analysis of communication identifies network patterns that are likely to contribute to a failure. Communication schedules containing those patterns are associated to failures. Developers use these associations to choose schedules for replay and inspection. We associate schedules by differentiating them against each other and correlating the differences with failed assertions in the executed code. Developers can also use these associations to decide whether a failure is likely related to non-determinism in the network communication schedule or not.

A communication scheduler replays a selected communication schedule in a live system. The scheduler shapes the exchange of data between the network and the application. As network events pass through it, the scheduler re-orders network events to match the selected schedule. Developers can use this to check whether a failure is caused by non-determinism in network event order. If a sporadic failure reliably occurs during repeated replays of a failing schedule, it is likely caused by patterns in the network event order. When developers know that a particular schedule causes a failure, they can use the scheduler to replay that schedule and gather additional execution data. This additional recording changes the runtime behavior of the application. This disturbance influences non-deterministic aspects of the execution, which, given deterministic servers, is

the order of network events. The scheduler re-orders those events to match the selected schedule, thus removing the disturbance.

We use the replay mechanism to connect network events with code locations. Developers can then move between the network event schedule and the code of the systems in a distributed application. We achieve this by recording calls during execution to create an execution trace. We save the stack information network events which is associated with the methods in the call tree.

With these features we target challenges that are inherent to distributed applications: non-determinism, shared resource access, and multiple languages. Non-determinism is intrinsic to unreliable networks, shared resources are required to persist shared data, and multiple languages are the reality of the Web. Developers need tools to approach these challenges, and, with our approach, they can investigate sporadic failures, compare and replay schedules, refine recorded state, and connect network events to code locations across languages.

Beyond debugging of communication, Replay-driven Fault Navigation allows developers to gain a larger picture of a distributed application. They can review communication patterns, and connect those patterns with specific code locations to understand how communication is performed in the code. Furthermore, connecting code locations on different servers via network events emphasizes the implicit programming interfaces between servers.

We have conducted interviews with developers and a trial with one developer to evaluate our approach. The outcome supports our claim that our tool is useful to understand and inspect non-deterministic failures in distributed systems. However, to determine whether it is faster than existing approaches a proper user study should be conducted.

Our current implementation of Replay-driven Fault Navigation targets asynchronous request-based distributed applications which can be modeled with UML sequence diagrams. Communication in sequence diagram is assumed to take no time, and so it is in our approach. In some distributed applications requests can take a considerable amount of time, however, which means that while such a request is in transit, a number of other requests may be sent and received. An example for such applications are those that use streaming or optimistic protocols, such as Google's SPDY[1], to minimize latency. For our approach to work with such applications, we have to adjust it and treat the beginning and end of a request as separate events, instead of the request as a single, atomic event.

Other open questions remain. Our heuristics and visualizations thereof for differencing network schedules should be expanded and tested in user study. We

[1] SPDY: An experimental protocol for a faster web
http://www.chromium.org/spdy/spdy-whitepaper
accessed July 20, 2012

also need to extend our prototype for distributed systems other than Web applications to show the generality of our approach. We want to explore integration of our logging and replay mechanisms into the services of hosting providers. This would make debugging distributed applications more accessible for developers that use such platforms. Finally, we think that we could extend our approach to other kinds of communication non-determinism, such as between processes and threads.

Despite these open questions, our approach is already usable to debug failures in distributed systems. We have evaluated on a prototype how developers can approach failures without prior knowledge of the concrete distributed system. Compared to traditional debuggers, our approach does not require the system to stop, it allows debugging on the deployed system, and it allows navigation on both network and implementation level.

Bibliography

[1] Gautam Altekar and Ion Stoica. "ODR: output-deterministic replay for multicore debugging". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, Oct. 2009, pages 193–206.

[2] Malte Appeltauer. "Extending Context-oriented Programming to New Application Domains: Run-time Adaptation Support for Java". PhD thesis. Hasso-Plattner-Institute, Apr. 2012.

[3] Malte Appeltauer and Robert Hirschfeld. "Declarative layer composition in framework-based environments". In: *Proceedings of the Workshop on Context-oriented Programming*. ACM, June 2012, pages 1–6.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. "A Comparison of Context-oriented Programming Languages". In: *Proceedings of the Workshop on Context-oriented Programming*. ACM, July 2009, pages 1–6. ISBN: 9781605585383.

[5] Shay Artzi and Michael D Ernst. "ReCrash : Making Software Failures Reproducible by Preserving Object States". In: *Proceedings of the European Conference on Object-Oriented Programming*. Springer, July 2008, pages 542–565.

[6] Melinda-Carol Ballou. *Improving Software Quality to Drive Business Agility*. Technical report. IDC, June 2008, pages 1–12.

[7] John J. Barton and Jan Odvarko. "Dynamic and Graphical Web Page Breakpoints". In: *Proceedings of the 19th international conference on World wide web*. ACM, Apr. 2010, pages 81–90.

[8] Sanjay Bhansali, Wen-ke Chen, Stuart De Jong, Andrew Edwards, Ron Murray, Milenko Drinic, Darek Mihocka, and Joe Chau. "Framework for Instruction-level Tracing and Analysis of Program Executions". In: *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, June 2006, pages 154–163.

[9] Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and Jean. Utke, editors. *Advances in Automatic Differentiation*. First. Springer, July 2008.

[10] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules". In: *Communications of the ACM* (May 1966), pages 366–371.

[11] Marc L. Corliss. "Low-overhead Interactive Debugging via Dynamic Instrumentation with DISE". In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Feb. 2005, pages 303–314.

[12] Anne Dinning and Edith Schonberg. *An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection*. First. ACM, Mar. 1990.

[13] Stephane Ducasse, Lucas Renggli, David C Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. First. Square Bracket Associates, Apr. 2010.

[14] Engine Yard. *The state of PaaS: 2012*. Technical report. Engine Yard, 2012.

[15] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. First. Prentice Hall, Aug. 2005, pages 1–5.

[16] Jason Gait. "A Debugger for Concurrent Programs". In: *Software: Practice and Experience* (June 1985), pages 539–554.

[17] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. "Replay Debugging for Distributed Applications". In: *Proceedings of the annual conference on USENIX'06 Annual Technical Conference*. May 2006, page 27.

[18] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?" In: *In Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*. IEEE, Jan. 1986, pages 3–12.

[19] Thorsten Grötker, Ulrich Holtmann, Holger Keding, and Markus Wloka. *The Developer's Guide to Debugging*. Second. CreateSpace, Apr. 2012.

[20] Robert Hirschfeld and Pascal Costanza. "An Introduction to Context-oriented Programming with ContextS". In: *Generative and Transformational Techniques in Software Engineering (GTTSE) II*. July 2008, pages 396–407.

[21] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the future: the story of Squeak, a practical Smalltalk written in itself". In: *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 818. ACM, Oct. 1997, pages 318–326.

[22] Christopher G. Kaler, Oliver J. Sharp, Erik B. Christensen, Dale A. Woodford, and Luis Felipe Cabrera. *Debugging Distributed Applications*. 2007.

[23] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P Anderson. "Cost-benefit Analysis of Cloud Computing versus Desktop Grids". In: *Proceedings of the 23rd International Parallel and Distributed Processing Symposium*. IEEE, May 2009, pages 1–12.

[24] Ravi Konuru and H. Srinivasan. "Deterministic replay of distributed Java applications". In: *Proceedings of the 14th International Parallel and Distributed Processing Symposium*. IEEE, May 2000, pages 219–227. ISBN: 0-7695-0574-0. DOI: 10.1109/IPDPS.2000.845988.

[25] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* (July 1978), pages 558–565.

[26] Friedemann Mattern. "Virtual Time and Global States of Distributed Systems". In: *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Oct. 1988, pages 215–226.

[27] Giuliano Mega and Fabio Kon. "Debugging Distributed Object Applications With the Eclipse Platform". In: *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*. ACM, pages 42–46.

[28] Robert Charles Metzger. *Debugging by Thinking - A Multidisciplinary approach*. First. Elsevier Digital Press, Nov. 2003.

[29] Krishna Nadiminti and Rajkumar Buyya. "Distributed Systems and Recent Innovations: Challenges and Benefits". In: *InfoNet Magazine* (Mar. 2006), pages 1–5.

[30] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyon Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Transactions on Software Engineering* (May 1983), pages 240–247.

[31] Sérgio Almeida Paulo, Carlos Baquero, and Victor Fonte. "Interval Tree Clocks". In: *In Proceedings of 12th International Conference on Principles of Distributed Systems*. Springer, Dec. 2008, pages 259–274.

[32] Michael Perscheid, Michael Haupt, and Robert Hirschfeld. "Test-Driven Fault Navigation for Debugging Reproducible Failures". In: *Journal of the Japan Society for Software Science and Technology on Computer Software* (2012).

[33] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. "Immediacy through Interactivity: Online Analysis of Runtime Behavior". In: *Proceedings of the 17th Working Conference on Reverse Engineerings*. IEEE, Oct. 2010.

[34] Tobias Rho, Malte Appeltauer, Stephan Lerche, Armin B. Cremers, and Robert Hirschfeld. "A Context Management Infrastructure with Language Integration Support". In: *Proceedings of the Workshop on Context-oriented Programming*. ACM, July 2011, pages 3–8.

[35] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Second. Pearson Higher Education, July 2004.

[36] Yasushi Saito. "Jockey: A user-space library for record-replay debugging". In: *Proceedings of the 6th International Symposium on Automated Analysis-driven Debugging*. ACM, Sept. 2005, pages 69–76.

[37] Terry Stanley and Tyler Close. *Causeway: A message-oriented distributed debugger*. Technical report. HP Laboratories, 2009.

[38] Dennis Strein and Hans Kratz. "Design and Implementation of a high-level multi- language .NET Debugger". In: *In Proceedings of the 3rd International Conference on .NET Technologies*. May 2005, pages 57–64.

[39] Gregory Tassey. "The Economic Impacts of Inadequate Infrastructure for Software Testing". In: *National Institute of Standards and Technology, RTI Project* (May 2002).

[40] Aaron E Walsh. *UDDI, SOAP, and WSDL: The Web Services Specification Reference Book*. First. Pearson Education, Apr. 2002.

[41] Zhonghua Yang and Keith Duddy. "CORBA: A Platform for Distributed Object Computing". In: *SIGOPS Operating Systems Review* (Oct. 1996), pages 4–31.

[42] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Second. Morgan Kaufmann, 2009.

[43] Hubert Zimmermann. "OSI reference model–The ISO model of architecture for open systems interconnection". In: *IEEE Transactions on Communications* (Apr. 1980), pages 425–432.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst sowie keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe.

Potsdam, den 20. August 2015

Tim Felgentreff