Science of Computer Programming 134 (2017) 4–18

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Implementing record and refinement for debugging timing-dependent communication

T. Felgentreff*, M. Perscheid, R. Hirschfeld

*Software Architecture Group, Hasso-Plattner Institute, Prof.-Dr.-Helmert Str. 2-3, University of Potsdam, Germany*

ARTICLE INFO

*Article history:*
Received 1 March 2014
Received in revised form 9 October 2015
Accepted 13 November 2015
Available online 28 November 2015

*Keywords:*
Distributed debugging
Record and replay
Dynamic analysis
Record and refinement

ABSTRACT

Distributed applications are hard to debug because timing-dependent network communication is a source of non-deterministic behavior. Current approaches to debug non-deterministic failures include post-mortem debugging as well as record and replay. However, the first impairs system performance to gather data, whereas the latter requires developers to understand the timing-dependent communication at a lower level of abstraction than they develop at. Furthermore, both approaches require intrusive core library modifications to gather data from live systems.

In this paper, we present the Peek-At-Talk debugger for investigating non-deterministic failures with low overhead in a systematic, top-down method, with a particular focus on tool-building issues in the following areas: First, we show how our debugging framework Path Tools guides developers from failures to their root causes and gathers run-time data with low overhead. Second, we present Peek-At-Talk, an extension to our Path Tools framework to record non-deterministic communication and refine behavioral data that connects source code with network events. Finally, we scope changes to the core library to record network communication without impacting other network applications.



## 1. Introduction

An increasing number of companies develop distributed Web applications [1]. Such applications often use custom communication interfaces rather than middleware [2]. This leads to timing dependencies in network communication and possible non-deterministic failures because the order of network events can cause changes in system behavior [3].

Non-deterministic failures are hard to debug, because developers cannot reliably reproduce them [4]. Without reproduction, developers cannot systematically test their hypotheses about possible failure causes and debugging becomes a time-consuming trial and error approach. Standard symbolic debuggers do not allow developers to work backwards from a failure to a root cause. In addition, for distributed systems, they provide no high-level view on communication between processes and network nodes.

State-of-the-art approaches to debug non-deterministic failures use either expensive recording for post-mortem debugging [5] or hard to understand record-and-replay techniques [6]. Post-mortem debugging collects all run-time data during execution and presents the combined data from different processes, which developers can then inspect independently from the running system. Record-and-replay techniques only record the outcome of non-deterministic operations to reproduce a

* Corresponding author.
*E-mail addresses:* tim.felgentreff@hpi.uni-potsdam.de (T. Felgentreff), michael.perscheid@hpi.uni-potsdam.de (M. Perscheid), robert.hirschfeld@hpi.uni-potsdam.de (R. Hirschfeld).

http://dx.doi.org/10.1016/j.scico.2015.11.006
0167-6423/

specific failure. All non-deterministic operations such as certain system calls, access to uninitialized memory, and unsynchronized access to shared memory are replaced by their recorded outcomes and allow developers to replay the distributed system.

The aim of both approaches is to present recorded data for inspection. Post-mortem debuggers record data during one execution at comparatively high overhead, while record-and-replay approaches record only enough data to ensure future runs are equivalent, and then uses multiple executions to incrementally record more data on each subsequent run. However, both approaches have limitations for debugging distributed network applications:

(a) Post-mortem debugging imposes large overhead [5] because all required data has to be recorded beforehand. This makes it unfeasible to record all data at once. Moreover, the large performance impact of these approached increases the potential for Heisenbugs [4] in which the act of observing influences the non-determinism behavior.
(b) Record and replay provides the wrong level of abstraction for understanding timing-dependent communication. During replay, developers still have to use symbolic debuggers and infer high-level network timing dependencies from the source code of distributed components. The information that a certain network request arrived after another does not easily map to which methods get executed in which order.
(c) Neither approach allows developers to request additional data from the live system if required during debugging. If the recorded data was insufficient or the replay system is different than the live system in a certain aspect, information required to understand and fix the bug may not be available to the developers. They have to re-record all previous execution data plus the additionally required information for post-mortem debugging, or have to configure the replay system to be more like the live system.

To lower the overhead of post-mortem debugging and guide developers to failure causes, we have proposed *test-driven fault navigation* [7], implemented in our Path Tools framework. Using test cases as entry points to reproducible behavior, Path Tools integrate anomaly detection [8,9] into a systematic breadth-first search for tracing failure causes back to defects. Our *incremental dynamic analysis* ensures a feeling of immediacy when debugging by splitting the analysis over multiple executions. Unfortunately, a key requirement for this approach is that failures must be deterministic.

To debug non-deterministic, timing-dependent network communication, we have propose an approach we call *record and refinement*. We implemented this approach as an extension to the Path Tools framework called Peek-At-Talk [10] and generalized it from there. Our implementation is particularly suited for scenarios where changes in the order of network events trigger a failure. The approach relies on recording network communication schedules and analyzing the differences between executions to detect anomalies that are likely causes of the non-deterministic failure. We then constrain the distributed system to the failing schedules by modifying its network communication analogous to a traffic shaper [11]. If the constrained system reproduces the failure repeatedly, developers can request and refine more data from the live system with the help of Path Tools.

To selectively modify the live system for our incremental recording and analysis, we describe a technique to transparently enable core library instrumentation. Our approach applies context-oriented programming to scope tracing of network communication for the debugging session, without influencing other, unrelated network applications.

Our combined approach imposes low overhead, provides a top-down view on timing-dependent communication, and still allows developers to inspect run-time data from the live system during debugging. In this paper, we describe the implementation of these techniques in our prototype Peek-At-Talk and its integration with our Path Tools framework. The contributions of this approach are:

- A combination of communication recording and anomaly detection techniques help developers identify failing communication schedules.
- A novel use of traffic shaping in conjunction with the incremental analysis provided by test-driven fault-navigation allows developers to split the recording and refinement of debugging data over multiple runs, thus imposing low overhead on the system and allowing debugging in the live system.

The contributions with respect to our tool's implementation in Squeak/Smalltalk [12] are:

- An architectural overview of our Path Tools framework that provides the debugging tools used for our incremental refinement and test-driven fault navigation.
- The description of our Peek-At-Talk debugger that implements our record and refinement approach, and which is integrated into the Path Tools framework and extends it for debugging non-deterministic failures.

Our approach is heavily driven by a concrete non-deterministic failure that we experienced in a distributed application and how we extended our Path Tools framework to debug this kind of failure. The original application is not open source, but we describe a smaller application that exhibits a similar bug in Section 2. Because this failure was difficult to debug using standard tools, we show how we extended our existing test-driven fault navigation approach (Section 3) with Peek-At-Talk, illustrated in Section 4. From this practical implementations, we distill the general components for record and refinement (Section 5). Section 6 presents related work and Section 7 concludes.

## 2. Background

We introduce a motivating example and the underlying system that serve as a basis for our debugging tools in the following sections.

### 2.1. Motivating example: timing-dependent flight booking

We present an example application, a flight booking service that uses asynchronous communication between servers. It includes: *a*) a Web server that allows customers to search for and book flights, *b*) a cache that the Web server communicates with to get flight information, and *c*) a crawler that collects flight information from different airlines and stores it in the cache. The flight information, such as available seats and pricing, changes over time and the cache periodically updates its contents.

In this system, a non-deterministic failure can occur, for example, under the following circumstance: two customers access the Web page at the same time, looking for flights from Berlin to Amsterdam on the same date. If both customers try to reserve seats on the same flight at roughly the same time, the booking is influenced by: *a*) the latency of their connection to the Web server, *b*) the cache update interval, and *c*) the changing flight information. It may happen that one of the two customers is able to book the flight at the displayed rate, while the other can only receive a higher rate, because now fewer seats are available and the crawler updated the cache. The less fortunate customer may then complain in a bug report like this:

> 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 was displayed in the list when I clicked it.[1]

The above bug report contains the word “sometimes”, indicating a degree of non-determinism. Such failures are not reliably reproducible and they are hard to observe by debugging tools because their analysis changes the corresponding failing behavior. As developers cannot systematically test their hypotheses and follow failure causes back to defects, non-deterministic failures are considered to be the most difficult ones [4].

### 2.2. Squeak/Smalltalk: debugging in a self-sustaining environment

This failure originates in our Squeak/Smalltalk environment [12] and we first tried debugging with its standard tools. Squeak is an open source implementation of the dynamic object-oriented programming language Smalltalk which only consists of objects. It offers a comprehensive development environment including a symbolic debugger, a flexible and extensible user interface, and rich meta-programming facilities. Squeak is almost entirely written in Smalltalk. This gives us extensive access to implementation details so that we can easily analyze, debug, and change the entire system. Furthermore, its self-sustaining environment makes it possible to enrich arbitrary objects with additional information such as their execution in network communications and to preserve this knowledge for later purposes. Even if Squeak already provides more debugging features than other development environments, it still faces several challenges with respect to localizing failure causes.

Standard symbolic debuggers as are available for most programming languages (with the exception only of exotic or experimental languages) are not well-suited for localizing failure causes. They suffer from missing advice on what is going wrong and back-in-time capabilities. A symbolic debugger only allows developers to stop a program and to access the run-time stack at a particular point in time. After that, they can experiment with the system only in the forward direction even though the defect is located in the past. To make things worse, the breakpoint and stepping functionality strongly influence non-deterministic failures by changing their timing behavior. A probable outcome of this is that these failures do not occur while observing them. All these shortcomings lead rather to disorganized trial and error debugging sessions than a systematic procedure for localizing failure causes.

## 3. Test-driven fault navigation for debugging reproducible failures

In order to solve the challenges of standard debugging tools, we introduced our test-driven fault navigation approach and its corresponding Path Tools framework [7]. Based on reproduced failures in form of test cases, we integrate anomaly detection [8,9] into a systematic breadth-first search for tracing failure causes back to defects. We implemented this debugging guide in our Path Tools framework including an extended test runner and a lightweight back-in-time debugger. In this section, we discuss our approach and its implementation as a basis for the Peek-At-Talk extension for debugging non-deterministic failures.

[1] We provide a simplified implementation of this service, including tests and the tools described in this paper as a stand-alone SHARE [13] VM at https://is.ieis.tue.nl/staff/pvgorp/share/.

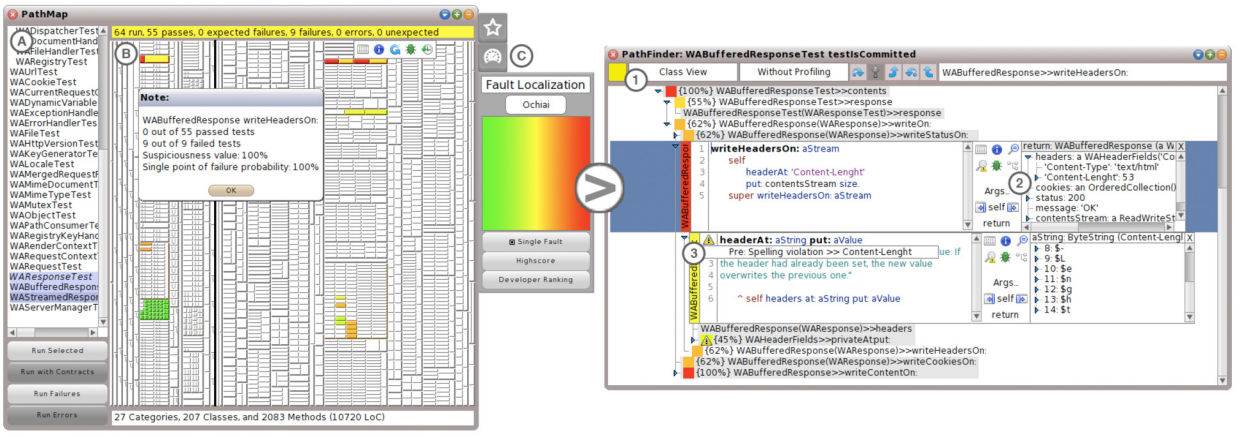


**Fig. 1.** PathMap as an extended test runner and PathFinder as a lightweight back-in-time debugger implement our test-driven fault navigation. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

### 3.1. Reproduce motivating example as test case

As a precondition for our test-driven fault navigation, developers have to reproduce the observable failure in the form of at least one test case. Our Path Tools suite analyze these test cases to localize and guide developers to failure causes. In the case of our non-deterministic motivating example, a test case that ("sometimes") reproduces the failure looks like follows:

```
testBugReport001
  | priceTable flightIdIdx flightId price reservation |
  priceTable := (HTTPSocket
                  httpGetDocument: 'http://localhost:4567/search'
                  args: {'origin' –> {'Berlin'}.
                         'destination' –> {'Amsterdam'}.
                         'date' –> {Date today printString}}) content.
  price         := (table
                    copyFrom: (table indexOfSubCollection: '<td>' startingAt: priceEndIdx – 9)
                    to: (table indexOfSubCollection: ' EUR')) asInteger.
  flightIdIdx := table content indexOfSubCollection: 'data–id="'.
  flightId      := (table content copyFrom: idIdx to: idIdx + 20) asInteger.
  reservation := (HTTPSocket
                  httpPostDocument: 'http://localhost:4567/flights/', id, '/reserve'
                  args: nil) content
  self assert: (reservation endsWith: price asString).
```

The purpose of this script is to create the Hypertext Transfer Protocol (HTTP) traffic that might trigger the failure. It mimics the behavior of a customer who reserves a seat on an airplane, communicates with the Web server via HTTP, and asserts that the returned data has the correct rates. We use assertions to classify recorded network communication schedules in relation to whether they produce a failure.

### 3.2. Test-driven fault navigation

To debug such difficult failures, our *test-driven fault navigation* [7] provides developers with a debugging guide. This guide interconnects multiple light-weight debugging tools into a framework that supports complex debugging scenarios and guides developers through from tool to tool. To do so, Path Tools applies anomaly detection [8,9] to find likely failure causes and uses a breadth-first search for tracing failure causes back to defects, starting at a high abstraction level and guiding the developer to the low level defective code that caused the failure. To do so, Path Tools starts with at least one test case that reproduces the observable failure and a number of test cases that show desired behavior, localizes anomalies by comparing behavior and state of all failed and passed test cases, and provides entry points for debugging. To find anomalies, we are looking for methods that are executed by failing tests cases, but not (or only rarely), by passing tests. These methods have a higher failure cause probability (anomaly) than methods being executed by less failing but many passing test cases.

We implement our test-driven fault navigation as part of the Path Tools framework[2] for the Squeak development environment [12]. The tool suite as presented in Fig. 1 mainly consists of our enhanced test runner PathMap [14] and our lightweight back-in-time debugger PathFinder [15]. PathMap is an extended unit test runner that does not only verify test cases but also localizes failure causes. Its integral components are from left to right a testing control panel (A), a compact tree map visualization of the software system (B), and several flaps for accessing various analysis techniques (C). Especially,

[2] http://www.hpi.uni-potsdam.de/hirschfeld/trac/SqueakCommunityProjects/wiki/pathToolsFramework.

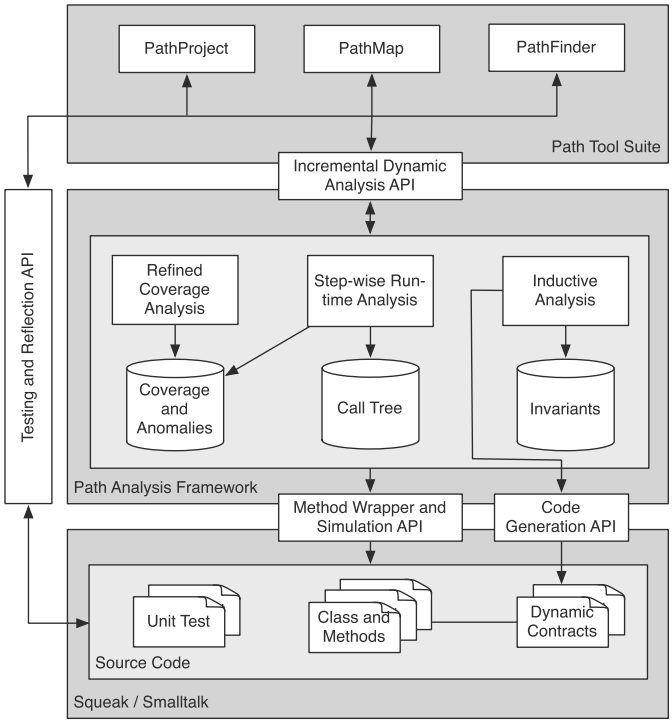


**Fig. 2.** The Path Tools framework is integrated into the Squeak/Smalltalk development environment and consists of a dynamic analysis framework and our tool suite.

the flaps on the right (C) set PathMap into specific analysis modes for collecting valuable feedback during the execution of test cases. With the activated *fault localization* flap, we automatically record method coverage of test cases, compute spectrum-based anomalies [8], and color the tree map (B) with suspiciousness and confidence scores (the more red the higher the failure cause probability). PathFinder is our lightweight back-in-time debugger for exploring specific test case executions with a special focus on fault localization. Not only does it provide immediate access to run-time information [15], but also classifies traces with suspicious behavior [16] and anomalous state [17]. Its main components are a control panel on the top (1) and the test case execution history in form of a call tree (2) below. From top to bottom, each node represents one method call and their subtrees describe its called methods. Developers can follow traces in both directions and explore more details on demand. After indicating interest in a specific argument, receiver, or return object, we reexecute the test case, make a deep copy of the requested object, and present it in an object explorer on the right (2). To integrate anomalies into the execution history, we reuse the colors from PathMap and map errors into the call tree by adding small exclamation marks (3) to methods. Finally, the Path analysis framework provides the basis for our tools on top of the SUnit testing framework[3] and implements our *incremental dynamic analysis* that ensure a feeling of immediacy when debugging with our approach [7]. Based on reproducible entry points such as test cases that always ensure the same execution path, this analysis interactively splits the expensive dynamic analyses over multiple runs. So, we can ensure a high degree of automation, scalability, and performance during debugging with our tools. For a more detailed description of our tools and how to debug with them, we refer to [18].

One important requirement of our test-driven fault navigation is the reproducibility of failures. Unfortunately, the bug report from above contains the word “sometimes”, indicating that the failure did not occur consistently, because of one or more sources of non-determinism in the system. Even if our Path Tools can partly work with such kinds of failures, the conditions are not ideal. For example, developers have to run PathMap multiple times until the failure occurs and PathFinder can only provide a shallow method call tree without state refinements. For that reason, we present in this paper the Peek-At-Talk tool that further allows developers to debug time-depending failures with our previous approach.

### 3.3. Architecture of the path tools framework

Before we present the extension to our Path Tools framework, we first introduce the existing architecture. Fig. 2 summarizes our Path Tools framework, consisting of 41 packages, 252 classes, 3307 methods, and 17,559 lines of code, with respect to the underlying Squeak/Smalltalk development environment.

[3] We consider unit test frameworks as a technique for implementing different kinds of test cases such as acceptance, integration, and module tests.

At the top, our *Path Tool suite* consists of the small helper tool PathProject, the extended test runner PathMap and the lightweight back-in-time debugger PathFinder. *PathProject* defines the scope of all further Path Tools and our incremental dynamic analysis. Therefore, this tool requires access to the source code in order to specify the system under observation. *PathMap* needs Smalltalk's testing application programming interface (API) to control the underlying unit test framework, the reflection API to determine a tree map of the system's structure, and the incremental dynamic analysis API to reveal anomalies. *PathFinder* applies the testing and reflection API to control specific test runs and to show source code in corresponding call trees. Such call trees are built with the help of our incremental dynamic analysis API that starts step-wise run-time analysis [15] and assigns anomalies.

In the middle of our architecture, the *Path analysis framework* supports the observation of unit test behavior by implementing our incremental dynamic analysis. The *refined coverage analysis* rapidly records the relationship between unit tests and executed methods and refines statement coverage of selected methods on demand by re-executing their corresponding test cases. With the help of this analysis, PathMap reveals behavior anomalies within a short amount of time and at different levels of detail. *Step-wise run-time analysis* divides the dynamic analysis of one specific test case over multiple runs. PathFinder applies this analysis to record a simple method call tree, highlight anomalies in the execution history, and refine additional behavioral information. We rely on method wrappers and Smalltalk's interpreter simulation for recording run-time information. At the level of methods, we collect run-time information with flexible method wrappers [19]: a wrapper introduces new behavior before and after the execution of a specific method without changing its original behavior. Depending on the chosen analysis technique, wrappers collect among others coverage, method calls, and state refinements. To record statements of a specific method, a special wrapper starts and stops Smalltalk's simulation engine that analyzes dedicated byte codes only. Both analysis techniques are necessary since a full simulation would slow down the execution by a factor of at least 100. Finally, the framework stores all collected measurements and makes this data available to any interested tool.

Our Path Tools framework only requires source code and unit tests from the Squeak/Smalltalk system. We access and analyze these two program artifacts with our three different APIs. First, the *testing and reflection* API is required by all Path Tools in order to access unit tests and the structure of the system. It controls test executions and introspects program entities. Second, the *method wrappers and simulation* API summarizes all meta-programming features for our incremental dynamic analysis. It offers a flexible way to implement arbitrary tracers and data structures that record run-time behavior such as call trees and invariants.

## 4. Peek-At-Talk – record and refinement in Squeak/Smalltalk

Recall the original bug report for the flight booking service failure (see Section 3.1). The failure occurred "sometimes", indicating some source of non-determinism. The original Path Tools design, however, relies on being able to reproduce failures reliably. In this section, we present an extension to Path Tools with *record and refinement* to debug non-deterministic failures on the network. Using our approach, developers can test whether communication timing dependencies are the source of this non-deterministic failure, using *communication recording*, *anomaly analysis*, and *refinement of remote data*.

In this section we will show, starting with the test script from Section 3.1, how a developer can use the Peek-At-Talk extension in Path Tools to execute the script a number of times. Peek-At-Talk will, during each execution, record a communication trace and our anomaly detection then analyses these traces and presents the developer with a selection of network events that are likely to be on the path to the failure. The developer can then, by selecting an anomalous event, trigger Peek-At-Talk to re-execute the script, this time enforcing the same order of network events and recording more data about the selected event. With that additional data, Peek-At-Talk determines the code that is executed around the selected network event, and thus guides the developer from the high level communication to the code, where the developer can then inspect program state to learn if and how the anomalous network event causes the failure.

For implementing Peek-At-Talk, we extend in particular the PathMap test runner and the PathFinder back-in-time debugger. First, we add diagnosis scripts as new entry points into reproducible behavior. We consider diagnosis scripts as separate from test cases, because their non-deterministic behavior requires an indeterminate number of executions to cause the failure. Second, we extended PathFinder with a pane for the network schedule and connect the call tree with the corresponding diagnosis scripts. Thus, with these small adaptations Peek-At-Talk has access to execution histories of network nodes and the Path Tools framework has been enlarged for debugging non-deterministic failures in distributed applications.

### 4.1. Communication recording

As for deterministic failures, in a distributed setting, developers also need to establish failure causes from their effects. To debug failures in any system, the order in which events occur is essential [20]. For a truly parallel system such as a system distributed over the network, we can create a partial order using *Lamport Clocks*. This order is called the *logical schedule* [21] in which the events occurred. Fig. 3 shows such a schedule for our example system. In this schedule, two requests arrive at the Web server. The first is sent on to the cache, which queries the crawler, and eventually returns the data to the Web server. Then the second Web request arrives at the cache, and is served from cached data. Upon receiving responses from the cache, the Web server responds to the Web clients.

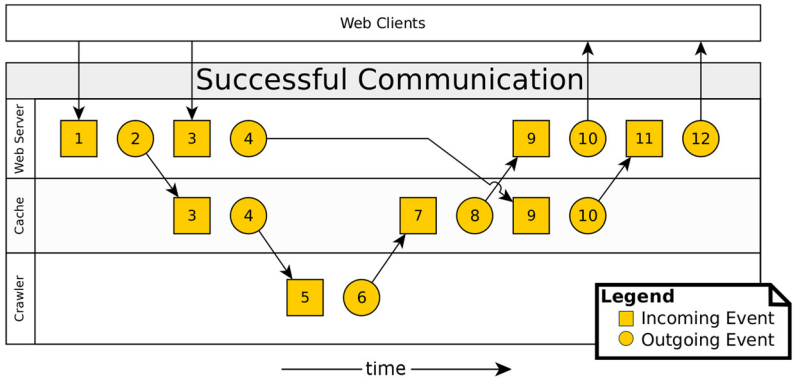


**Fig. 3.** A communication schedule.

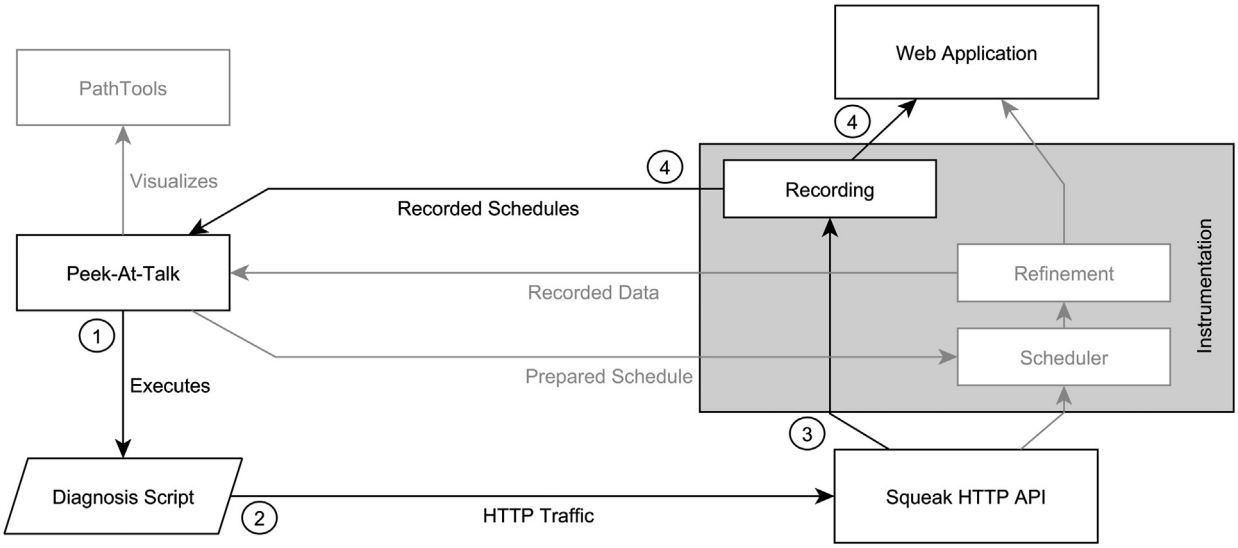


**Fig. 4.** Communication Recording Phase.

To be useful for debugging, the logical schedules we record have to be sufficiently detailed to capture differences across multiple executions. Assuming that servers are otherwise deterministic, we have to record order, direction, and communication partners for each network event in a distributed application. This can be done with little overhead both in space and time during the actual execution. The necessary information overhead we have to encode into each event consists of 8 byte Lamport clocks [6], with the first bit as a sign byte to tell incoming from outgoing events. Additionally, we include IP addresses and port numbers to identify the communication partners, which adds 4 bytes for the port and 16 bytes for the address, in the worst case of IPv6 addressing. For an application that keeps the last 1,000,000 requests stored in its logs, this amounts only to about 26 MiB in additional storage space. The additional time required to encode this information is negligible, inducing a slowdown of only 1% in our Peek-At-Talk, with most of the time spent waiting for network I/O operations. Fig. 4 shows how the different modules collaborate during recording.

The HTTPSocket class is responsible for handling HTTP traffic. We have instrumented this class to record a process-local trace of the network communication for each send and receive, before passing the traffic on. Network traces in Peek-At-Talk include an integer id as Lamport clock, the first 500 bytes of the message, as well as IP address, URL, and port for both sender and receiver for all sent and received HTTP requests. These are saved in a process-local structure and are not sent on the network. The recording layer also adds the current Lamport clock "time" as an application defined HTTP header to each outgoing request, and reads it from incoming requests. This is HTTP specific, and for protocols that do not have room for custom data, a more general approach – such as prepending magic bytes to the payload – can be used [21].

When developers run a diagnosis script through the PathMap tool (cf. Fig. 5①), Peek-At-Talk asks how often the script should be executed (this number has to be chosen from experience, finding a number of executions that shows around a dozen failures seems to work well – so for a script that fails 10% of the time, 100 runs are ok). After each recording, Peek-At-Talk merges the traces from different processes into a list, sorts events by id, and assigns the result of the test case to the trace. This way, each trace consists of a list of events that include communication partners and a Lamport clock, as well as a test result. Test results are gathered and shown in a banner in the user interface (UI) (Fig. 5②).

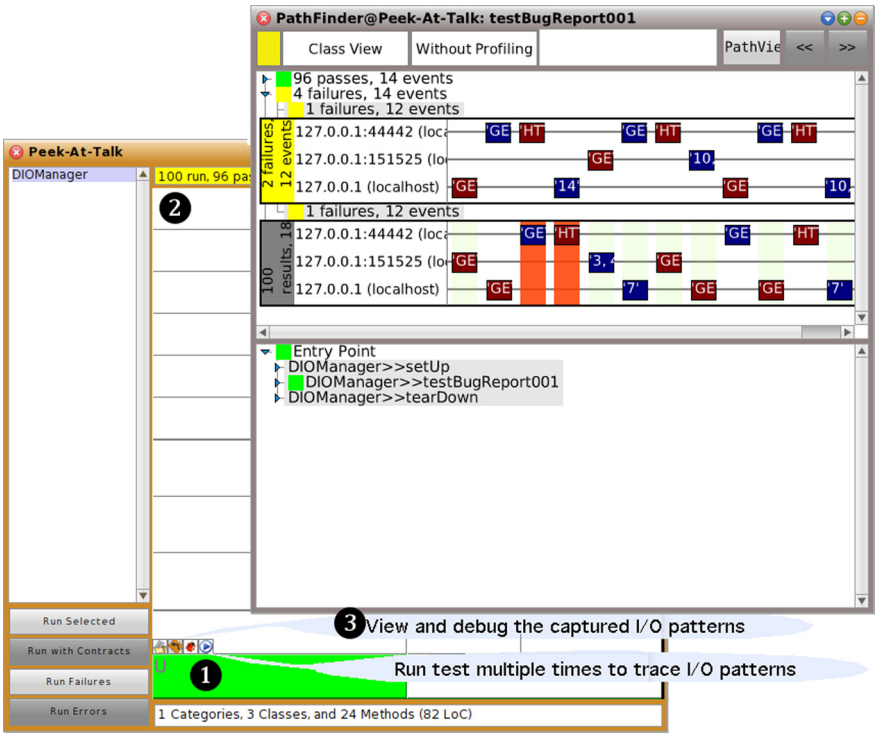


**Fig. 5.** Diagnosis scripts are run multiple times from the PathMap, and the PathFinder highlights anomalies between executions.

### 4.2. Anomaly detection

Variances between successful and failing communication schedules can be used to reduce the number of network events that developers need to consider during debugging. We employ the longest common subsequence (LCS) [22] differencing algorithm to groups of communication schedules to detect variances and highlight anomalies. First, we determine LCSs of all successes and all failures respectively, removing outliers. Second, we diff LCSs of successful communications with LCSs of failing communications. The differences between those are most likely to contribute to a failure.

For more complex the communication patterns, anomaly detection requires more recorded executions to detect anomalies. Assuming otherwise deterministic servers, successful schedules will exhibit fewer anomalies and thus a sufficiently large number of failing and successful executions will ensure that the anomaly detection can exclude variances that have little or no impact on the failure. If the servers exhibit other non-determinism, additional detection of such is required to discards unsuitable executions, e.g., if a load balancer is present, the test script has to fail if a different server is used.

Peek-At-Talk uses the PathFinder tool from the Path Tools suite to display anomalies. After data for a test case has been recorded, developers can select a test case to open the PathFinder tool. This tool shows a call tree for the selected test case and normally allows developers to examine the arguments and return values of each method in the execution history of a test. Peek-At-Talk extends PathFinder and shows a differencing view above the execution history. This view shows the difference between all runs, between the passing runs, and between the failures separately. Shown in Fig. 5 is the outcome for our example bug report: across all (passing and failing) runs, the first incoming and outgoing traffic for the topmost communication endpoint are identified as problematic. Developers should then begin their investigation with the anomalous events or events immediately surrounding them.

### 4.3. Refinement of remote data

If communication timing dependencies are the source of a non-deterministic failure, then the system will fail consistently when constrained to failing communication schedules. Once developers reviewed and understood the high-level dependencies, they need to understand how they are expressed at the source code level to fix the failure.

*Schedule selection* The first goal is to enable consistent reproduction of failures in the live system by removing the network as a source of non-determinism. From the data recorded during multiple executions, developers can select one of the failing schedules to constrain the system to this particular communication schedule. For this we implement a *traffic shaper* [11], which, on each node, mediates application access to the network. If the system repeatedly exhibits the failure when it is constrained to this communication schedule, non-determinism on the network is likely the culprit.

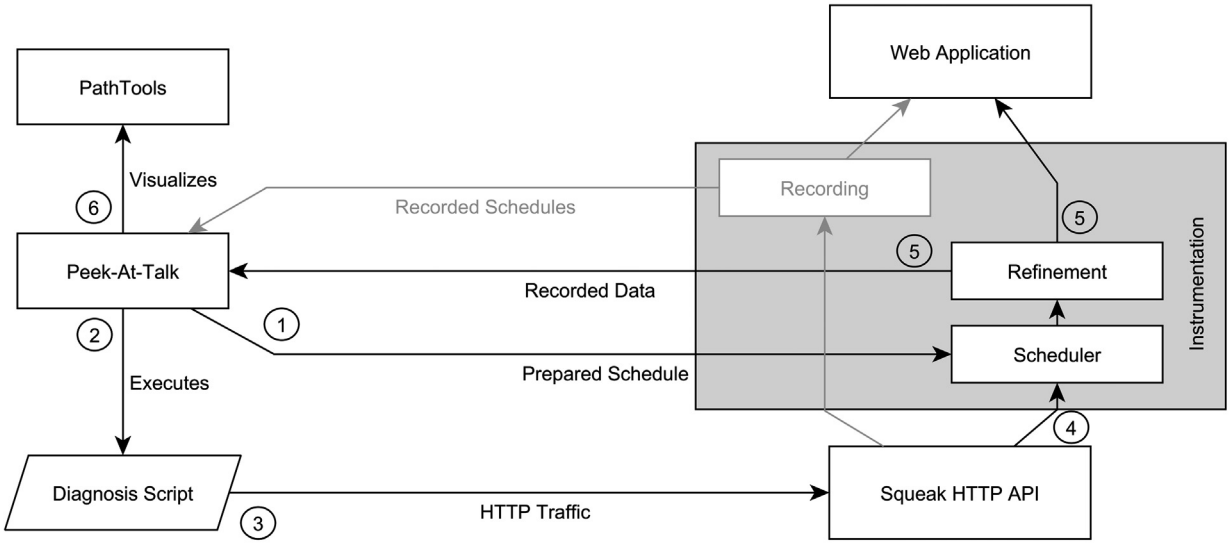


**Fig. 6.** Scheduling and Refinement.

The scheduler controls all network access by wrapping Squeak's network API. All data sent or received through that API is treated according to the recorded schedules. For this, the scheduler buffers each event. Once all events that are expected to occur before a buffered event have appeared according to the schedule, the buffered event is released. In the case of a single-threaded server, any interaction with the network is served from the buffer if possible, and only executes the wrapped functions if no suitable event is available. The scheduler will keep executing the wrapped function and add the result to its buffer until a suitable event is available. On a multi-threaded server, events are held back using semaphores that are signaled when the previous event is released.

Fig. 6 shows that the schedule is selected from Peek-At-Talk, and then the diagnosis script is re-run. The traffic shaper keeps a thread-safe input and output buffer to re-order network events as required.

Our scheduler has to deal with situations in which no event occurs that matches a required schedule. Such situations will cause an application to wait for the network and appear to "hang". In cases where a given schedule cannot be reproduced in the live system, for example if a server that appears in the schedule has been taken offline, we argue that developers have to be able to inspect the scheduler operations to understand the situation. Appropriate heuristics such as timeouts or conditions for giving up on a replay attempt are developer configurable.

*Call tree inspection* Using our scheduler, developers can inspect specific failing communication schedules for further analysis by expanding the anomaly detection view. Each differential schedule is the parent node of a tree of schedules from which it is the difference. Developers expand this tree and select any one of the failing schedules to re-run it. For each schedule, labels on the left hand side reports how many runs had this particular communication schedule, how many events are in this schedule, and which endpoints were encountered in the communication.

The first time a developer re-runs a schedule, Peek-At-Talk adds a query to record the stack at the time each event is released. The process that generates that event leverages the reflective capabilities of Smalltalk to gather execution data. When a schedule is applied to the live system, our scheduler records which methods are active before each event is released and returns that data to our debugging tool. A developer can then select an event in the communication view, and the debugging tool will present a partial call tree that corresponds to the selected event.

Using the existing Path Tools facilities, developers can then explore the call tree and request additional information about method arguments and return values, as well as instance variables of objects. Based on the recorded stack, Peek-At-Talk displays the method source code, if it is available. If developers try to inspect, for example, a method argument, PathFinder wraps the corresponding method on the on local system, and creates deep copies of its arguments and return values during the next re-execution, which are then returned to the debugging tool.[4]

In Fig. 7 we show the result of a developer clicking the event after the two problematic events from Fig. 5. If the source code for the displayed methods is available in the image running the debugger, the application method closest to the event automatically opens below. The first event (①) is colored in blue, which means it is an *incoming* event (probably the answer to the first outgoing request). Here, it is the answer to a request to the /overview/ URL of the data warehouse, as we can see from the code. Checking the next event, we find that an unrelated endpoint sends an update to the data warehouse.

[4] For copying objects, we rely on Smalltalk's `veryDeepCopy` protocol, which relieves us of the burden of dealing with details of copying. If necessary, this protocol also allows developers to adapt the analyzed object depth to their needs.

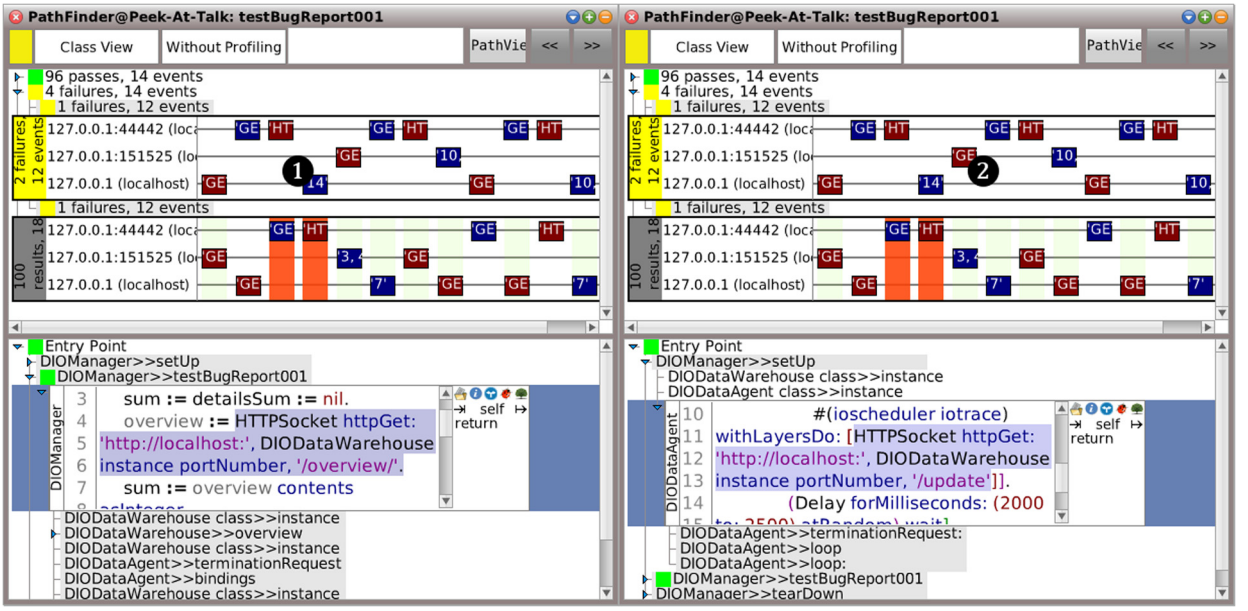


**Fig. 7.** Refinement events in the network schedule (top) to show the corresponding application methods (bottom). (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

Such an update may have invalidated the data the test script received in the overview. The defect then might be that the client requests details from an overview that has been invalidated by an intermediate update.

To check this hypothesis, developers can then select method arguments or receiver instance variables for inspection. This approach is an extension of step-wise run-time analysis [15]. Developers can follow the information available in the system until they understand the failure, without causing a significant one-time overhead. The overhead of each exploration step is small, because we only selectively record information that developers request.

### *4.4. Scoping core library modifications*

For non-deterministic failures in a distributed setting, observations from the live system are essential [23], because long-running systems will over time accumulate state changes and encounter situations unforeseen during development [6]. Thus, we record information in the live system, including information about executions that exhibit the failure.

However, we are interested only in the communication that occurs when a diagnosis script is executed against the live system, not all communication at all times. More specifically, we would like to separately enable and disable all three modifications – recording, traffic shaping, and refinement – we made to the core HTTP library. Thus, we want our modifications to be scoped to particular executions, instead patching the core library globally.

To scope our changes, we instrument Smalltalk's distributed API, e.g. HTTP, with *ContextS* [24], a context-oriented programming (COP) [25] implementation for Squeak. COP is an extension to object-oriented programming that provides a *layer* construct for encapsulating cross-cutting behavioral variations of a system. Layers can adapt classes and methods and can be composed at run-time. COP implementations vary in their limitations based on the underlying system, but for Squeak, *ContextS* can wrap any method.

Many dynamic languages – object-oriented languages like Ruby and Smalltalk, but also Scheme-like languages – allow developers to override or extend core functionality through a technique dubbed "monkey-patching". However, we have found COP to be a better option for developing tools that need to adapt core parts of the programming environment and language for two reasons. First, the changes to the core system are dynamically scoped, so we can constrain the adaption to our tool only. This way, as Lincke et al. have found [26], we do not have to worry if tools may rely on subtleties of the original implementation – they will not see our adaptions. Second, layers as a separate entity can be independently source-controlled – that means that we were able to continuously update our adaptions in the running system without breaking existing running instances.

For Peek-At-Talk, we have created Context/S layers for recording and traffic shaping. Activation of each layer is completely transparent to users debugging a diagnosis script through Peek-At-Talk. Our layers adapt all methods that read and write HTTP requests to and from a socket. This includes both communication triggered directly from Peek-At-Talk, as well as communication received from a system that is running Peek-At-Talk. In the first case, the layer is enabled using the normal dynamic scoping of COP, for the second case, the layer is enabled when a custom HTTP header field is set, as described in service-oriented applications of COP [27]. This implementation approach can be applied analogously to other APIs as well as in other programming languages for which a COP [25] mechanism is available.

**Table 1**
Executions of a script that exercises the described flight booking failure. We ran the script 50s and averaged the number of executions per 5s.

| Benchmark | Avg executions/5s | $\sigma$ |
|---|---|---|
| Normal execution | 8.51 | 0.31 |
| Recording | 8.39 | 0.12 |

Although ContextS has a very high-overhead and is one of the slowest available implementations of COP [25] for send-heavy benchmarks, we consider its performance adequate. The benchmark results in Table 1 present the overhead of our recording layer. In the case of distributed applications, a lot of the time is spent waiting for I/O and not in method activation. Given these results, a context-oriented implementation for other languages should equally have a very small practical overhead and thus fulfill the requirement for low-overhead tracing.

## 5. Distilled concepts of record and refinement

We argue that our approach can easily be adapted to other object-oriented programming languages. We have embedded our prototype in the Path Tools framework, showing that it is not necessary to support an entire back-in-time debugger to implement our approach. The Path Tools framework only relies on some dynamic analysis techniques, that can be easily implemented, for example, with aspect-oriented programming [28] or reflection capabilities of the host language. For Peek-At-Talk, we used COP to scope recording and traffic shaping in core libraries. This language extension is available for a range of programming languages [25], and thus an implementation in another programming system could use a similar approach.

### 5.1. Identifying entry points for debugging

Our implementation uses a script similar to a test case, which we call a *diagnosis script*. More generally, any program that acts as a client in the distributed system and that can be run over a longer period of time while the participating nodes are recording network events is suitable for our approach. One has to ensure that the execution of the script itself does not disrupt the system so much as to cause a denial of service, for example, by adding a delay between script executions. As the script is running, the participating nodes record only the most basic information to recreate the order of messages that was sent.

To aid developers establish causal relationships between different program states, events, and subsequent failures, we have implemented logical schedules for this in Peek-At-Talk (cf. Section 4.1), and we argue that this is the most suited approach at communication recording, given that it is still widely used [21,5,6,23] and extension such as Vector Clocks [29], Version Vectors [30], and Interval Tree Clocks [31] exist for more specific scenarios.

Note that our approach only works under the assumption 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. This may preclude usage of our approach for less dependable networks.

We correlate events from the live system over multiple runs to help developers understand the reasons why one particular log ends with a failure while another exhibits none, and reduce the entry points into the distributed program that the developer has to check. We find that sequence diagrams are well suited to model network schedules as a *call tree*, with requests and responses representing asynchronous message sends and returns. To compare these network schedules, we have used the LCS algorithm in our prototype. While it seems to give good results, we lack experience with the approach to fully recommend it, and alternative differencing algorithms should be explored.

In the implementation, we also weigh differences in communication schedules according to the following heuristic, where $N$ is a function that maps from network events to sets of traces that include them, $P$ is a function that maps from network events to sets of passing traces that include them, and $W$ is a function that assigns each event a probability from 0 to 1 to be good entry point for debugging.

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

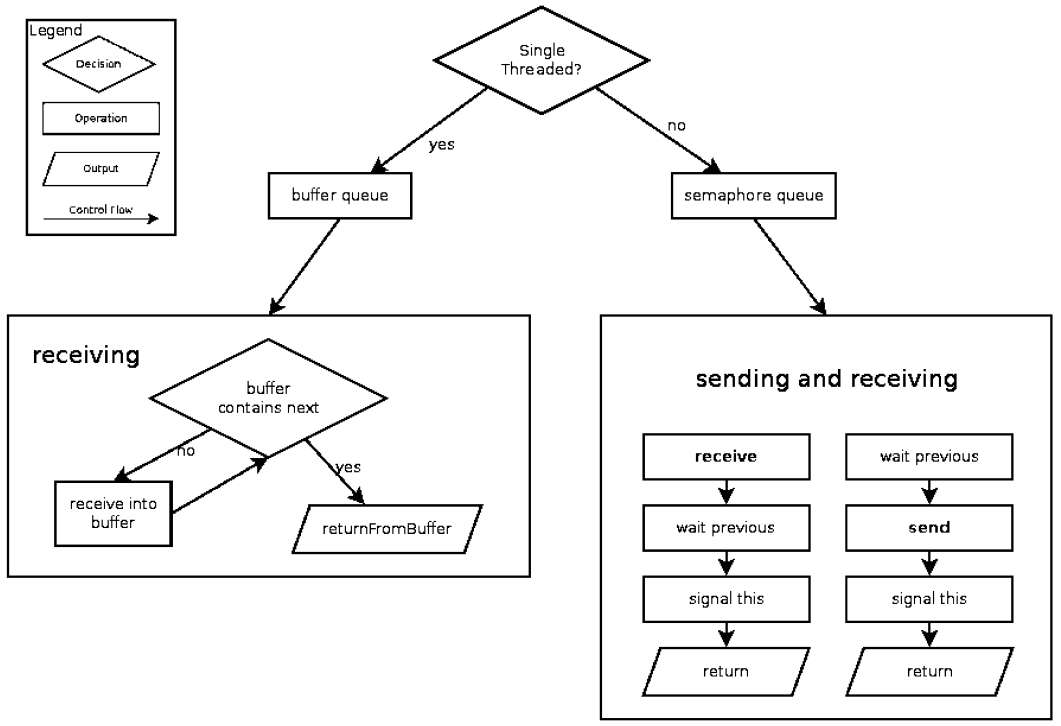


**Fig. 8.** FlowChart [32] of Re-ordering Scheduler.

The heuristic considers interesting sequences to be critical regions in the communication schedule, where even slight variations may cause a failure. These are typically sequences that vary little between successful runs, but are different for failing runs. The variations that occur within successful runs can be considered robust regions of communication, where variance has little or no impact on the successful execution. If we compare successful schedules against the failing schedules, and subtract the variances within the successful schedules themselves, we can 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. When failing schedules have variances within themselves, we can further use this to mark these variances as less likely causes than others. 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. 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. The main issue with our heuristic is that it needs a large number of test runs with a sufficient number of both passes and failures.

### *5.2. Incremental refinement of distributed debugging data*

Although our approach is useful to find timing-dependencies in any setting, regardless of external clients, it is most useful in the closed world case. The final piece to our approach is the combination of *traffic shaping* [11] with incremental recording and refinement of debugging data. Traffic shaping is a commonly used approach to influence network communication, and our work recording and refinement in the context of test-driven fault navigation [16] is combined here to constrain the communication in the system to failing communication schedules. The traffic shaping is implemented in Peek-At-Talk as the *scheduler*, inserted between the application framework code responsible for sending and receiving, and the underlying language or operating system primitives.

The use of a traffic shaper means that our approach can only be used in scenarios where all participants that were active during recording are active during replay as well. A practical implementation should communicate to the developers when this is not the case, so the developer can take appropriate action, such as choosing another schedule or ensuring the required servers are indeed online. Generally, such a traffic shaper 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 Fig. 8. Such a scheduler will have to distinguish 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.

If communication timing dependencies are the source of a non-deterministic failure, then the system will fail consistently when constrained to failing communication schedules. Once developers reviewed and understood the high-level dependencies, they need to understand how they are expressed at the source code level to fix the failure. To that end, they can now use the approach of test-driven fault-navigation to record more debugging data. The only requirement here is that the test-driven fault-navigation is again available in all nodes that should be inspected, and that they allow access to record further data through a network-transparent debugging interface like Java Debugger Interface (JDI).

## 6. Related work

Our approach builds on previous work in the area of distributed post-mortem debugging and distributed record and replay. Current approaches vary mainly in how much they record, and subsequently, how much of the system they can simulate to eliminate behavioral disturbance.

We draw ideas for our debugging tool and refinements approach from Friday [33], a debugger for distributed record and replay, and Causeway [5], a distributed post-mortem debugger. Friday provides extensions to the GDB debugger and enables developers to add watchpoints and queries to a replay to gather more information. Causeway synchronizes a network view with a call-tree, so developers can easily map executed code to network events. In contrast to these approaches, our record and refinement enables developers to continuously use it on the live system to debug failures that occur very rarely or never in development.

Our recording technique is similar to the record and replay as presented in liblog [6], DejaVu [21], RecPlay [34], and Jockey [35], all of which also use Lamport clocks and intercept I/O to establish partial ordering of network events. However, these approaches focus on low-level network events and intercept I/O at the level of systems calls. They also require either a custom virtual machine (VM) or additional dynamic libraries that cannot be deactivated at run-time. The low-level events recorded by these approaches are further removed from the application, which makes it difficult for developers to understand where they originate in their application code. As the kind of data that is recorded by these approaches is predetermined, developers cannot easily test whether a given non-deterministic failure originates from timing dependencies in network access. Compared to these approaches, record and refinement is more flexible in what it records, and does so at a higher level abstraction. The additional guidance provided in the Path Tools frameworks helps developers connect high-level network events with low-level code on the different servers.

R2 [36] is an application-level record-and-replay approach which offers more control to the developers about what is recorded, and can be used to test for multiple sources of non-deterministic failures. However, unlike our approach, R2 does not allow replay in the live system.

The limited recording approach we take can also be seen in relation to scalable back-in-time debugging techniques [37, 38]. However, those scalable techniques often rely on infrastructure to record program traces efficiently, and even if they are efficient enough to record in the live system (as is our approach), in our approach we are looking specifically at rare, non-deterministic failures. Since these occur only very rarely, the data that has to be recorded for these back-in-time debugging techniques can still grow very large, requiring extra storage on each node in the network. With our approach, on the other hand, the recorded communication schedules and debugging data on each run is small, with additional runs used to record additional data slices. These small data slices can be immediately transferred to the debugging machine, and storage requirements on the distributed servers are limited.

Our record and refinement can be seen as an extension of dynamic memory access race-detection in parallel programs such as RaceTrack [39]. Compared to such approaches our implementation is less general. However, its advantage is the higher level of abstraction, which is closer to the mental model of the Web application developer. We argue that tools that detect a large set of data-races, but do so by acting at a low level of abstraction, such as memory regions and I/O devices, are less useful in debugging specific kinds of races than specialized tools such as Peek-At-Talk.

## 7. Conclusions

We have presented our Path Tools framework for test-driven fault navigation, and its extension Peek-At-Talk for *record and refinement*. Our tools guide developers with a systematic top-down method to debug non-deterministic failures in the timing-dependent communication of distributed applications. The tools provide low-overhead recording and anomaly analysis of network events in the live system, and refining of run-time data during debugging.

Since the overhead of our recording approach is low, it can be enabled in a deployed system continuously and record data about failures that occur infrequently or only during deployment. We achieve this low overhead by recording only the data strictly necessary to reproduce a failure, and by scoping the core library modifications so that only relevant parts of the system are instrumented.

Our anomaly analysis of communication schedules identifies network patterns that are likely to contribute to a failure. Developers use these associations to acquire an overview of the timing dependencies in the system at a high level of abstraction and subsequently to choose schedules and events for further inspection at the source code level. We detect anomalies by differentiating communication schedules and correlating the differences with failed assertions.

The refinement mechanism of Path Tools is re-used in Peek-At-Talk to selectively inspect run-time data for a particular communication schedule. This allows developers to debug the failure at the source code level, without being restricted by the amount of prerecorded information. We achieve this by constraining the system to a particular communication schedule and wrapping method executions to record run-time data.

The presented approach currently has two main limitations. First, our low-overhead recording and schedule shaper depend on the determinism of each server. Consequently, if other sources of non-determinism, such as system calls to `random` or `gettimeofday`, have an impact on the failure, we cannot currently reproduce it. This can be fixed by recording more sources of non-determinism, at the expense of increased overhead. This can be easily implemented with additional layered methods around such non-determinism. However, our scheduler could still use that additional data to constrain the live system to reliably reproduce a failure, and the rest of our approach still applies.

Second, our approach currently only works in the *closed-world* case, in which all processes that participate in the distributed system include our recording and scheduler layers. If that is not the case, we need to extend our approach in two ways to support an *open-world* case: first, we need to use a discovery mechanism to tell participating processes from non-participating processes. Second, for non-participating processes, we need to record the message contents besides what we currently record. If a schedule is later applied to the live system, events from and to non-participating systems are replayed from the recorded data. Note that this may not work if data returned from non-participating servers is time-dependent, in which case our approach is not applicable.

Future work thus is two-fold. First, recording and scheduling should be extended to other sources of non-determinism, to show that our approach is applicable not only to timing-dependent communication. Second, we are investigating how to integrate non-participating servers into our approach. Furthermore, beyond debugging of timing-dependent communication, we may investigate how Peek-At-Talk allows developers to gain an overview of a distributed system. By connecting code locations on different servers via network events it emphasizes the implicit programming interfaces between servers and helps developers to understand how communication is performed in the code.

Despite these avenues for future work, our approach is already usable to debug failures in distributed systems. We have evaluated in a small case study with five developers how they used Peek-At-Talk to approach failures without prior knowledge of the concrete distributed system. Our preliminary findings indicate that, while the tool is of limited use for examples with very few network events and communication partners, it is more helpful to understand traces with hundreds of events in long running applications. This was expected, because our heuristic anomaly analysis works better when more variance is observed in successfully network communications. Our users reported that for large traces, the anomalous communication event that helped them to understand and fix a bug was usually clearly among the highest ranked, whereas for small traces, the ranking was often less clear. Nonetheless, our approach could be used as an additional tool to debug live systems, because it does not require the system to stop, it allows recording data directly from the instrumented live system with little overhead, and it allows navigation from the network events to the implementation level.

## Appendix A. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.scico.2015.11.006.

## References

[1] D. Kondo, B. Javadi, P. Malecot, F. Cappello, D.P. Anderson, Cost-benefit analysis of cloud computing versus desktop grids, in: IPDPS, IEEE, 2009, pp. 1–12.
[2] K. Nadiminti, R. Buyya, Distributed systems and recent innovations: challenges and benefits, InfoNet Mag. (2006) 1–5.
[3] C. Artho, K. Havelund, A. Biere, High-level data races, Softw. Test. Verif. Reliab. (2003) 207–227.
[4] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2009.
[5] T. Stanley, T. Close, Causeway: a message-oriented distributed debugger, Technical Report, HP Laboratories, HP Laboratories, 2009.
[6] D. Geels, G. Altekar, S. Shenker, I. Stoica, Replay debugging for distributed applications, in: ATC, USENIX, 2006, pp. 289–300.
[7] M. Perscheid, Test-driven fault navigation for debugging reproducible failures, Ph.D. thesis, University of Potsdam, 2013.
[8] J. Jones, M. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: ICSE, 2002, pp. 467–477.
[9] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, C. Xiao, The Daikon system for dynamic detection of likely invariants, Sci. Comput. Program. 69 (2007) 35–45.
[10] T. Felgentreff, M. Perscheid, R. Hirschfeld, Constraining timing-dependent communication for debugging non-deterministic failures, in: WASDeTT, 2013.
[11] IETF RFC 2475, Internet standard definition of “Shaper”, 1998.
[12] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: the story of squeak, a practical smalltalk written in itself, in: OOPSLA, ACM, 1997, pp. 318–326.
[13] P. Van Gorp, S. Mazanek, SHARE: a web portal for creating and sharing executable research papers, Proc. Comput. Sci. 4 (2011) 589–597.
[14] M. Perscheid, D. Cassou, R. Hirschfeld, Test quality feedback – improving effectivity and efficiency of unit testing, in: C5, IEEE, 2012, pp. 60–67.

[15] M. Perscheid, B. Steinert, R. Hirschfeld, F. Geller, M. Haupt, Immediacy through interactivity: online analysis of run-time behavior, in: WCRE, IEEE, 2010, pp. 77–86.
[16] M. Perscheid, M. Haupt, R. Hirschfeld, Test-driven fault navigation for debugging reproducible failures, J. Jpn. Soc. Softw. Sci. Technol. Comput. Softw. (2012) 188–211.
[17] M. Perscheid, T. Felgentreff, R. Hirschfeld, Follow the path: debugging state anomalies along execution histories, in: CSMR-WCRE, IEEE, 2014, pp. 124–133.
[18] M. Perscheid, R. Hirschfeld, Follow the path: debugging tools for test-driven fault navigation, in: CSMR-WCRE Tool Demo Track, IEEE, 2014, pp. 446–449.
[19] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: ECOOP, Springer, 1998, pp. 396–417.
[20] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM (1978) 558–565.
[21] R. Konuru, H. Srinivasan, Deterministic replay of distributed Java applications, in: IPDPS, IEEE, 2000, pp. 219–227.
[22] C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008.
[23] G. Altekar, I. Stoica, ODR: Output-Deterministic Replay for multicore debugging, in: SOSP, ACM, 2009, pp. 193–206.
[24] R. Hirschfeld, P. Costanza, An introduction to context-oriented programming with contexts, in: GTTSE, Springer, 2008, pp. 396–407.
[25] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A comparison of context-oriented programming languages, in: COP, ACM, 2009, pp. 1–6.
[26] J. Lincke, R. Hirschfeld, Scoping changes in self-supporting development environments using context-oriented programming, in: COP, ACM, 2012, pp. 2–10.
[27] M. Appeltauer, R. Hirschfeld, J. Lincke, Declarative layer composition with the JCop programming language, J. Object Technol. 12 (2013) 1–37.
[28] T. Gschwind, J. Oberleitner, Improving dynamic data analysis with aspect-oriented programming, in: CSMR, IEEE Computer Society, 2003, pp. 259–268.
[29] F. Mattern, Virtual time and global states of distributed systems, in: Proceedings of the International Workshop on Parallel and Distributed Algorithms, 1988, pp. 215–226.
[30] D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, C. Kline, Detection of mutual inconsistency in distributed systems, IEEE Trans. Softw. Eng. (1983) 240–247.
[31] S.A. Paulo, C. Baquero, V. Fonte, Interval tree clocks, in: Proceedings of 12th International Conference on Principles of Distributed Systems, Springer, 2008, pp. 259–274.
[32] C. Böhm, G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, Commun. ACM (1966) 366–371.
[33] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, I. Stoica, Friday: global comprehension for distributed replay, in: NSDI, USENIX, 2007, pp. 1–24.
[34] M. Ronsse, K. de Bosschere, RecPlay: a fully integrated practical record/replay system, ACM Trans. Comput. Syst. (1999) 133–152.
[35] Y. Saito, Jockey: a user-space library for record–replay debugging, in: AADEBUG, ACM, 2005, pp. 69–76.
[36] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M.F. Kaashoek, Z. Zhang, R2: an application-level kernel for record and replay, in: OSDI, USENIX, 2008, pp. 193–208.
[37] G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, SIGPLAN Not. 42 (2007) 535–552.
[38] A. Lienhard, T. Gîrba, O. Nierstrasz, Practical object-oriented back-in-time debugging, in: Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP'08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 592–615.
[39] Y. Yu, T. Rodeheffer, W. Chen, RaceTrack: efficient detection of data race conditions via adaptive tracking, in: OSR, ACM, 2005, pp. 221–234.