Constraining Timing-dependent Communication for Debugging Non-deterministic Failures

T. Felgentreff[a], M. Perscheid[a], R. Hirschfeld[a]

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

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 on.

In this paper, we present the Peek-At-Talk debugger for investigating non-deterministic failures with low overhead in a systematic, top-down method: first, we record network communication schedules with less performance overhead. Second, we detect anomalies in the network communication to help developers understand timing dependencies. Finally, we constrain subsequent communication in the live system to reliably reproduce failures and allow developers to inspect additional runtime data.

Keywords: Distributed Debugging, Record and Replay, Dynamic Analysis

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 are also inappropriate to debug distributed systems because they work only at the

Email 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)

source code level and provide no high-level view on communication between processes and network nodes.

More 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 runtime 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 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 debugging records during one execution at comparatively high overhead, while record-and-replay records only enough data to ensure future runs are equivalent, and then uses multiple executions to record a little more data each time. However, both approaches have limitations for debugging distributed network applications:

(a) Post-mortem debugging imposes a high overhead on live systems [5] because all required data has to be recorded beforehand. This overhead makes it unfeasible to leave recording enabled for the live system at all times. Moreover, these approaches have a potential to introduce 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.

We propose *record and refinement*—implemented in our Peek-At-Talk tool—to debug timing-dependent network communication. Our 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.

During the initial record phase, we record network communication in the live system and divide communication schedules into failing and successful schedules.

We analyze the differences between these schedules to detect anomalies that are likely causes of the non-deterministic failure. We constrain the distributed system to the failing schedules by modifying its network communication analogous to a traffic shaper [7] that inspects network requests and passes them on according to some rule. Our rule is the previously recorded schedule, so we remove timing-dependent communication as a source of non-determinism by delivering and sending requests in order. If the constrained system reproduces the failure repeatedly, developers can request and refine more data from the live system with the help of our extended step-wise run-time analysis [8]. The contributions of this paper are:

- A recording technique with low overhead that removes timing-dependent network communication as a source of non-determinism, and enables constraining communication in the live system so multiple equivalent executions are possible.
- A communication anomaly analysis that allows developers to approach a non-deterministic failure from the abstraction of network communication, in addition to the low-level source code abstraction.
- An approach to record runtime data over multiple executions to provide as much real data from the live system as required to understand the non-deterministic failure.
- The Peek-At-Talk tool that implements our record and refinement approach for Squeak/Smalltalk [9].

The rest of this paper is structured as follows: Section 2 illustrates our approach with the help of an example. Section 3 explains Peek-At-Talk and gives an overview of its implementation. Section 4 presents related work and Section 5 concludes.

2. Record and Refinement

We present an example system, 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.

This report contains the word "sometimes", indicating that the failure did not occur consistently, because of one or more sources of non-determinism in the system (one such source may be variations in user behavior, which we will ignore). We now present record and refinement with the help of this example.

2.1. Communication Recording

Developers could try to reproduce the problem on a development system first, but for non-deterministic failures, observations from the live system are essential [10]. Distributed systems are set up differently during development than at deployment time, and will over time accumulate state changes and encounter situations unforeseen during development [6]. Thus, instead of trying to reproduce the non-deterministic failure locally, we record information in the live system, including information about executions that exhibit the failure.

Given the above bug report, developers create hypotheses, which they need to test, about the failure's root cause. Using our approach, developers can test whether communication timing dependencies are the source of this non-deterministic failure. If so, they can inspect how communication varies over multiple executions to understand the failure top-down, before debugging at the source code level.

Developers need to establish causes from effects to debug failures in any system, and to do so, the order in which events occur is essential [11]. However, distributed systems are capable of truly parallel execution, in which case a global ordering is not obtainable. Instead, we establish a partial ordering of communication events through the use of *Lamport Clocks*. This order is called the *logical schedule* [12] in which the events occurred. Figure 1 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.

To be useful for debugging, the communication schedules we record have to be sufficiently detailed to capture differences across multiple executions. We assume that servers are otherwise deterministic, which allows us to limit the amount of information we need to record to communication events. Consequently, to record the causes of non-deterministic failures that stem from timing-dependent communication, we have to record order, direction, and communication partners for each network event in a distributed application. This can be

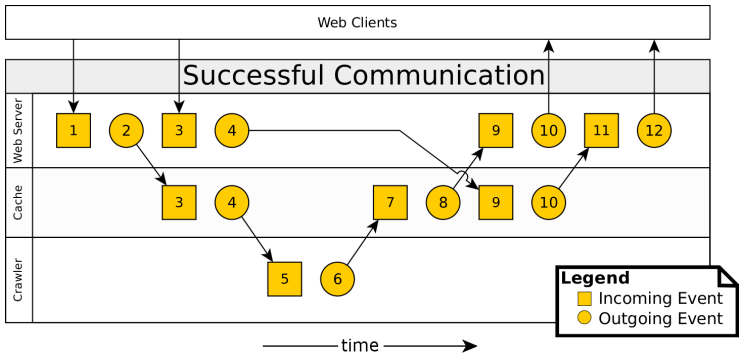

Figure 1: A communication schedule

done with little overhead both in space and time during the actual execution. The necessary information overhead we 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 before rotation, this amounts 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 Peek-At-Talk, with most of the time spent waiting for network I/O operations.

Benchmark	Avg Executions/5s	σ
Normal execution	8.51	0.31
Recording	8.39	0.12

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

Schedule Shaper. The first goal of record and refinement 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, we constrain the system to those communication schedules that exhibit the failure. For this we use a *traffic shaper* [7], which, for each node, controls the interaction with I/O devices and mediates application access to the network.

The scheduler has to control all network access by applications. We achieve this control by wrapping the runtime's network application programming interface (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.

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.

2.2. Anomaly Detection

To determine whether network races are the culprit of a failure, we require developers to write a *diagnosis script*. Such a script is similar to a test case:

```
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).
```

In our example, the script mimics the behavior of a customer who reserves a seat on an airplane. It communicates with the Web server via the Hypertext Transfer Protocol (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. Variances between successful and failing communication schedules can be used to reduce the number of network events that developers need to consider during debugging.

For our anomaly detection, we repeatedly apply the longest common subsequence (LCS) [13] differencing algorithm to groups of communication schedules. 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. In Figure 2, we have only one successful run at the top and one failing run at the bottom left, so the LCSs of failures and successes are trivially to the full communication schedules each. We compare those and find the differences between successes and failures in the shaded and magnified area.

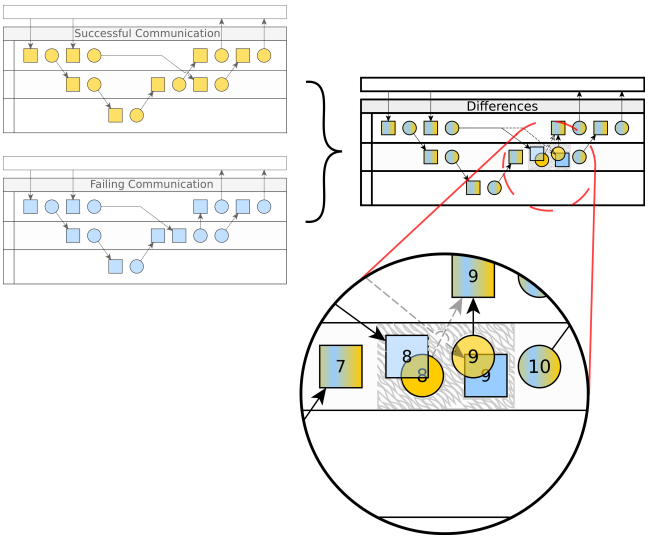

Figure 2: Anomaly detection for two schedules

For more complex examples, 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.

2.3. 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.

Using our scheduler, developers can inspect specific failing communication schedules for further analysis. The scheduler provides an interface to add queries to events. 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. When this 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 (cf. Section 2.3).

Developers can 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, our

tool attaches a query to the scheduler in the live system. The scheduler wraps the corresponding method and creates deep copies of its arguments and return values during the next re-execution, which are then returned to the debugging tool[1]. Developers can drill into object references analogously.

This approach is an extension of step-wise runtime analysis [8]. Developers can follow the information available in the live 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.

2.4. Discussion

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.

Secondly, 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. 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.

3. Implementation

We implemented Peek-At-Talk in Squeak/Smalltalk [9]. It works for distributed applications that use HTTP to communicate. We wrapped the Squeak HTTP API with *ContextS* [14], a context-oriented programming (COP) [15] 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 runtime.

[1]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.

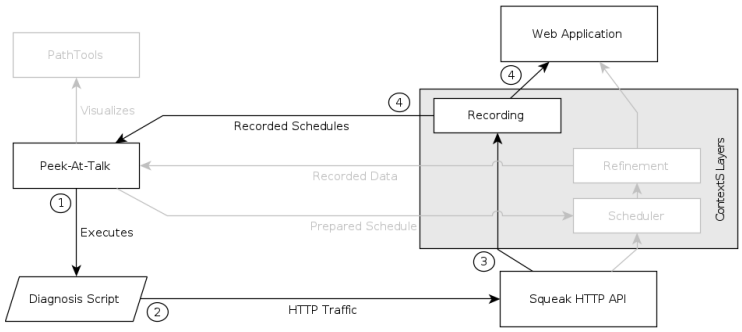

Figure 3: Communication Recording Phase

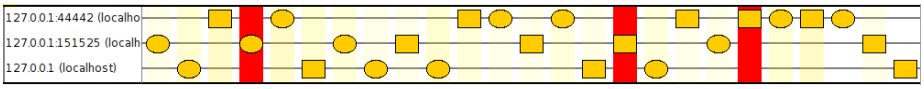

Figure 4: Highlighted anomalies

With ContextS layers we can dynamically enable recording and scheduling in the live system. The layers adapt all methods that read and write HTTP requests to and from a socket. This implementation approach can be applied analogously in other programming languages for which a COP [15] mechanism is available.

Communication Recording. Figure 3 shows how the different modules collaborate during recording. A diagnosis script creates HTTP traffic and the recording layer creates a process-local trace of the network communication before passing the traffic on to the application. Network traces 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. This layer also adds the current Lamport clock "time" as an application defined HTTP header to each outgoing request. This is HTTP specific, and for protocols that do not have room for custom data, another approach—such as prepending magic bytes to the payload—can be used [12]. After recording, we merge the traces from different processes and sort events by id in Peek-At-Talk.

Anomaly Detection. For anomaly detection, Peek-At-Talk requires developers to write a diagnosis script which can be executed to gather communication schedules. If the script includes assertions that check for the presence of a failure, Peek-At-Talk automatically compares and highlights anomalies of the failures as shown in Figure 4. Developers can then begin their investigation with these anomalous events.

Refinement of Remote Data. The scheduler layer, when active, constrains the system to a particular communication schedule. Figure 5 shows that the schedule is selected from Peek-At-Talk, and then the diagnosis script is re-run. The

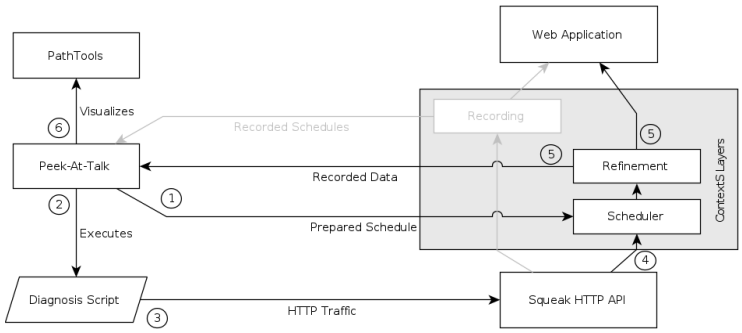

Figure 5: Scheduling and Refinement

schedule layer keeps a thread-safe input and output buffer to re-order network events as required. A schedule contains a recorded trace with queries to the refinement layer. Developers can inspect an event more closely by selecting it graphically, and Peek-At-Talk will activate the scheduler layers for each process with the corresponding schedule and queries. The first time a developer selects an event, a query is added to record method invocations around it. The process that generates that event leverages the reflective capabilities of Smalltalk to gather execution data and return it to the Peek-At-Talk debugger, which combines recorded method invocations and visualizes them in a call tree in *Path Tools* [8, 16]. In Figure 6 we show the result of a developer clicking event ①. 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. Developers can then select method arguments or receiver instance variables for inspection, which causes other queries to execute.

4. 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 [17], 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 [12], RecPlay [18], and Jockey [19], all of which also use

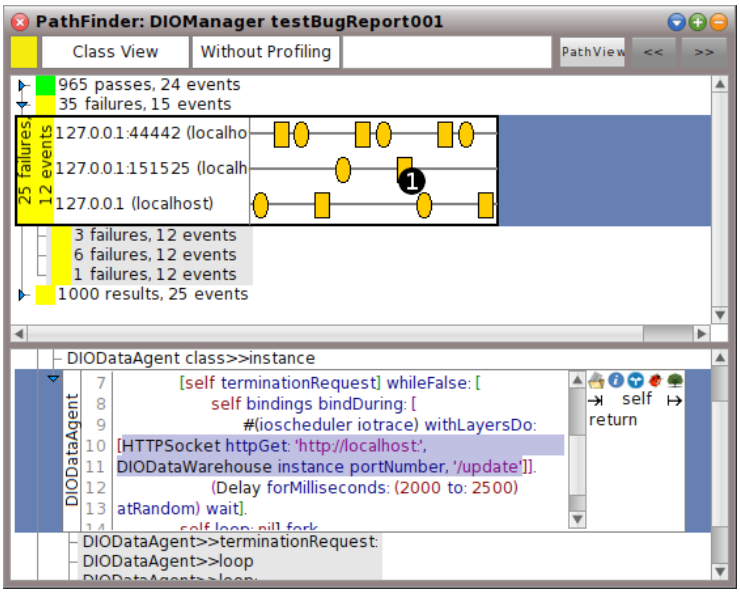

Figure 6: Refinement of an event in the network schedule (top) to show the corresponding application method (bottom)

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 runtime. 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.

R2 [20] 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.

Our record and refinement can be seen as an extension of dynamic memory access race-detection in parallel programs such as RaceTrack [21]. 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.

5. Conclusions

We have presented our *record and refinement* approach to debug non-deterministic failures in the timing-dependent communication of distributed applications. Our approach provides a systematic top-down method to low-overhead recording of network events in the live system, anomaly analysis, and refining of runtime data from the live system. To evaluate our approach, we present our Peek-At-Talk tool for debugging distributed Web applications.

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 focusing on the data strictly necessary for failures caused by timing-dependent communication.

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. We detect anomalies by differentiating communication schedules and correlating the differences with failed assertions.

Finally, we provide a refinement mechanism to selectively inspect runtime data in the live system. 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 runtime data.

Future work 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 future work, our approach is already usable to debug failures in distributed systems. We have evaluated in a case study how developers can approach failures without prior knowledge of the concrete distributed system. Our approach does not require the system to stop, it allows debugging on the live system, and it allows navigation on both network and implementation level.

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 Magazine (2006) 1–5.

[3] C. Artho, K. Havelund, A. Biere, High-Level Data Races, Software Testing, Verification and Reliability (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, 2009.

[6] D. Geels, G. Altekar, S. Shenker, I. Stoica, Replay Debugging for Distributed Applications, in: ATC, USENIX, 2006, p. 289–300.

[7] IETF RFC 2475, Internet standard definition of "Shaper", 1998.

[8] 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.

[9] 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.

[10] G. Altekar, I. Stoica, ODR: Output-Deterministic Replay for Multicore Debugging, in: SOSP, ACM, 2009, pp. 193–206.

[11] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM (1978) 558–565.

[12] R. Konuru, H. Srinivasan, Deterministic Replay of Distributed Java Applications, in: IPDPS, IEEE, 2000, pp. 219–227.

[13] C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008.

[14] R. Hirschfeld, P. Costanza, An Introduction to Context-oriented Programming with ContextS, in: GTTSE, Springer, 2008, pp. 396–407.

[15] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, M. Perscheid, A Comparison of Context-oriented Programming Languages, in: COP, ACM, 2009, pp. 1–6.

[16] M. Perscheid, M. Haupt, R. Hirschfeld, Test-Driven Fault Navigation for Debugging Reproducible Failures, Journal of the Japan Society for Software Science and Technology on Computer Software (2012) 188–211.

[17] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, I. Stoica, Friday: Global Comprehension for Distributed Replay, in: NSDI, USENIX, 2007, pp. 1–24.

[18] M. Ronsse, K. de Bosschere, RecPlay : A Fully Integrated Practical Record / Replay System, ACM Transactions on Computer Systems (1999) 133–152.

[19] Y. Saito, Jockey: A User-Space Library for Record-Replay Debugging, in: AADEBUG, ACM, 2005, pp. 69–76.

[20] 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.

[21] Y. Yu, T. Rodeheffer, W. Chen, RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking, in: OSR, ACM, 2005, pp. 221–234.