Saarland University

Faculty of Mathematics and Computer Science

Department of Computer Science

Masterthesis

Pathfinder: Exploiting Inter-Thread Communication for Request Flow Instrumentation

submitted by

Nicolas Valentin Schäfer

submitted

Januar 2021

Reviewers

Dr. Jonathan Mace

Prof. Dr. Deepak Garg

Abstract

In complex distributed services, the control flow of requests spans multiple software components, distributed over many processes and machines. Traditional machine-centric approaches, such as profilers and debuggers, cannot capture a coherent view on requests in such systems. To address this, special purpose tools arose to capture the end-to-end flow of requests. They all rely on propagating a *request context* (e.g. request ID) alongside the request across all components. This *context propagation* does not exist out-of-the-box. System developers must expend significant effort modifying their system to provide context propagation. This instrumentation process is difficult. Often, developers miss locations that require instrumentation; identify incorrect locations; or insert instrumentation incorrectly. We present Pathfinder, a developer-in-the-loop tool that guides the developer through the end-to-end instrumentation process. Pathfinder uses *runtime analysis* to detect potential instrumentation points of a system's source code, then directs developer attention to those places. Developers can then decide which of the detected points are indeed places that require instrumentation. Pathfinder focuses on intra-process context propagation and exploits inter-thread communication events to identify potential instrumentation points. Pathfinder was able to identify *all* intra-process instrumentation points in two complex, widely used distributed systems. With the help of Pathfinder less than 1% of each system's code base needed to be investigated manually to achieve a full system instrumentation.

Acknowledgements

None of this work would have been possible without the constant support of my advisor Jonathan Mace. His excellent feedback kept me on track. I would also like to thank Arpan Gujarati, Thomas Davidson and Vaastav Anand for their advices to improve this thesis. Finally, I would like to thank my family and Fabienne for encouraging and supporting me throughout the whole process.

Contents

Chapter 1

Introduction

In complex distributed services, the control flow of requests spans multiple software components, distributed over many processes and machines. In this setting it is significantly difficult to troubleshoot performance and correctness issues. For example, if the latency of a request is higher than average, it is difficult to deduce the root cause – i.e., what caused the request slowdown? – because doing so requires a coherent end-to-end view of the request. That is, we would need information about what happened during the request in *all* of the components, processes, and machines traversed by the request. To deduce the root cause, we would want to know the execution time in each component, or important events that may have happened on different machines.

Traditional machine-centric approaches, such as profilers and debuggers, cannot capture such a coherent view on requests in distributed systems. Consequently, special-purpose tools have been proposed by recent research, such as distributed tracing [1, 2], resource tracking frameworks [3, 4], and metric tracking systems [5, 6, 7]. All of these tools work at the granularity of *end-to-end requests*. In particular, to overcome the above challenges, all of these tools are based on *request flow tracking*.

Request flow tracking is a technique for capturing end-to-end information about requests in distributed systems. A simple example of request flow tracking is to use and propagate a unique *request ID* with each request. When each request arrives at the system, it is assigned a unique request ID. The system then includes this request ID with the request, everywhere it goes. If the request ID is present everywhere the request goes, performance and debugging information – such as events that happened during the request's execution, and measurements of execution times in different components – can include the ID of the "current" request. Afterwards, this information can be collected from all machines and grouped by request ID to reveal the end-to-end picture of the request's execution.

In general, request flow tracking relies on *context propagation*. A *context* – in the previous example, the request ID; in general, request-specific metadata – must be *propagated* alongside the request everywhere the request goes. Request flow tracking is difficult to achieve in practice because context propagation is challenging to implement correctly, as we show in §3.2. Context propagation must *correctly* pass contexts within and across *all* components, otherwise the system might fail to attribute work correctly to the right requests. Context propagation must be correct both *between* processes (e.g. contexts must be correctly included in RPCs) and *within* processes (e.g. contexts must correctly move between threads, through request queues, etc.).

Context propagation does not exist out-of-the-box. Instead, system developers must expend significant effort modifying their system to provide context propagation. Consequently, many distributed systems today lack context propagation, and it is a laborious process to deploy any tool (e.g. distributed tracing) that requires it. Although some recent work has explored automating context propagation through source code transformation and by inferring request paths [8, 9, 10], all prior approaches are incomplete and most are impractical. Consequently, context propagation today is still primarily a *manual instrumentation* task that must be performed by developers.

Manual instrumentation requires developers to extensively modify the source code of their system to identify places where contexts (e.g., request IDs) need to be propagated. For example, the context must be included in inter-process messages when sent, and extracted from those messages when received. Context is propagated across thread pools, event systems and custom execution infrastructures to correctly associate work done by each thread with the corresponding request. The developer must consider large parts of the code base and its dependencies to uncover all of these places where contexts must be propagated.

Unfortunately, instrumentation is challenging [11, 12, 13]. Developers must mentally understand the complete

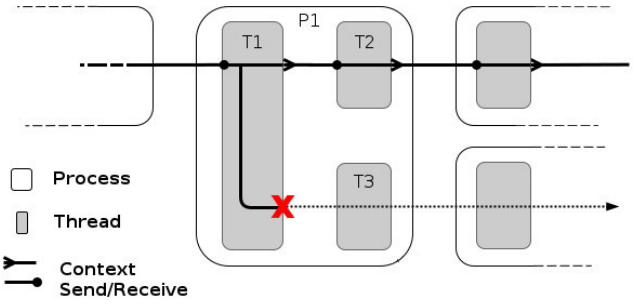

Figure 1.1: Incomplete context propagation along a request execution path. Because instrumentation is missing, T1 does not propagate the context to T3.

execution model of the system, map this understanding to specific locations in the program's source code, and finally correctly insert context propagation code at those locations. Often, developers miss locations that require instrumentation; identify incorrect locations; or insert instrumentation incorrectly [14, 15, 16, 17, 18, 19, 20, 21]. In all cases, this leads to incorrect request flow tracking: work will be incorrectly attributed to the wrong request, and often, work will not be attributed to a request when it should be. Moreover, as we discuss in §2.3 and §3.2, context propagation is sometimes use-case dependent and requires the developer to make qualitative decisions.

To illustrate why incorrect context propagation is a challenge, consider the example in Figure 1.1. In this example, a request enters process P1 and begins executing in thread T1. Part way through execution, T1 offloads some work to two other threads, T2 and T3; those threads then make remote calls to other processes. To correctly track the request, instrumentation is required in two important places: when work is passed from T1 to T2; and when work is passed from T1 to T3. If instrumentation is present, as illustrated for T1 to T2, then the work done in T2 and in subsequent processes can be correctly attributed to the request. However, if instrumentation is *missing*, as illustrated for T1 to T3, then *none* of the work will be attributed to the request, neither in T3, nor in any of the subsequent processes. Completeness of instrumentation is therefore critically important to correctly track request flows, as a single "missing link" can have an outsized effect.

The goal of this thesis is to make it easier to manually instrument systems. Specifically, we wish to reduce the developer effort and cognitive burden required to achieve complete and correct system instrumentation. Although instrumentation is primarily a manual task, we know of no existing tools to aid developers in instrumenting their systems. Our solution, *Pathfinder*, is a developer-in-the-loop tool that guides the developer through the end-to-end instrumentation process. Pathfinder uses *runtime analysis* to detect potential instrumentation points of a system's source code, then directs developer attention to those places. Developers can then decide which of the detected points are indeed places that require instrumentation. Pathfinder substantially reduces the cognitive burden on developers, because developers only need to consider a vastly-reduced set of potential instrumentation points.

Pathfinder is effective in identifying *intra-process* propagation points, widely considered to be the most difficult and pernicious kind to get correct [10, 22], and neglected by prior automatic instrumentation techniques [23, 24, 10]. Pathfinder is a general-purpose tool for Java programs. It is intended to be used at development or testing time by developers. Developers run their system with Pathfinder attached; Pathfinder will analyze the system as it runs, to detect potential instrumentation points. Developers can then update the system's source code with instrumentation. Pathfinder is general-purpose: it can capture any intra-process request flows including asynchronous event execution, when work items are submitted to thread pools, and when executions for different requests are paused and interleaved.

Pathfinder's key observation is that at instrumentation points there is usually an *information exchange* between the thread that currently executes the request and another that will continue the execution. For example, a common pattern we observe is that one thread creates objects to represent some work or event item, then passes that object to a different thread to perform the work. Pathfinder automatically tracks these information exchanges at runtime, and combines this information to present to developers as potential instrumentation points. The developer inspects this information, decides on relevance, and ultimately inserts instrumentation based on the information.

We evaluate Pathfinder by fully instrumenting two large distributed systems, HBase [25] and Cassandra [26]. With the help of Pathfinder we realized a full system instrumentation for both systems. We were able to identify *all* instrumentation points by investigating less than 1% of the code base of HBase and Cassandra respectively. We defined the ground truth of instrumentation points based on existing instrumentation both systems provide.

Our overall contributions are as follows:

- We identify *inter-thread communication* as highly discriminitive in determining instrumentation points.
- We develop an exhaustive method for detecting inter thread communications in program executions.
- We present a systematic developer-in-the-loop instrumentation workflow that takes advantage of tracked inter-thread communication events to guide the developer.
- We design and implement *Pathfinder*, an instrumentation tool, and present evaluation on two large open-source distributed systems.

The rest of this thesis is organized as follows. Chapter 2 introduces background on request tracking, the instrumentation process for context propagation and related work. Chapter 4 presents the concepts behind Pathfinder, including a mechanism to track inter-thread communication in program runs and a systematic instrumentation workflow. Chapter 6 presents the implementation of our Java prototype. Chapter 7 covers the evaluation of Pathfinder on two software systems. In Chapter 8 we discuss the results of the evaluation and future work. In Chapter 9 we conclude the thesis.

Chapter 2

Background

In this chapter, we present background and an overview on *request flow tracking*, which the work of this thesis then builds upon.

2.1 Distributed Systems

Modern internet services like Netflix [27] and YouTube [28] are distributed systems. They comprise multiple software components, which run in different runtimes and are distributed across many machines in a datacenter. This is necessary for the enormous scale of the applications and the performance and availability that they require. This contrasts with more traditional application designs which appear as single-process, monolithic systems.

The prevailing design approach for building distributed services is the *micro-service architecture*. Microservices isolate the different software components of an application into smaller, fine-grained pieces. They communicate with each other via runtime-independent protocols (e.g. remote procedure calls), which minimizes inter-service dependencies and offers much higher modularity.

Using a microservice architecture has several benefits. It enables teams of developers to work independently, using the software stack that best fits their project. Moreover, it enables independent evolution: the developers of one service do not need to know the internal details of other services.

However, the downside is an increase in system complexity. When a top-level request from a user enters the system (e.g. to load the user's personalized Twitter stream), the resulting execution can be an adventure across many different machines and software components. It will touch potentially hundreds of services and traverse many machines in the datacenter. Different requests follow different paths through the application, touch different machines, and utilize different resources. For example, in September 2019 Uber [29] reported over 3000 distinct microservices, where requests touch hundreds of nodes.

At any point in time, there may be millions of requests concurrently traversing the system. Within each service, there may be multiple requests queued and multiple requests executing concurrently. Every service may utilize complex scheduling and load-balancing mechanisms to ensure the system is efficient and robust under load, further affecting the paths each individual request may take.

2.2 Troubleshooting & Monitoring Distributed Systems

Because of the large and repeatedly changing set of machines a request execution traverses, it is difficult for system engineers to troubleshoot performance and correctness issues. For example, if the latency of a request is higher than average, the root cause may originate from *any* of the components, processes, or machines traversed by the request. To deduce this root cause – i.e., what caused the request slowdown? – requires the engineer to observe *everything* the request did during its end-to-end execution.

Many of the traditional tools and techniques that we have for debugging and diagnosing problems don't work well in this setting, because they are machine-centric. For example, logging, profilers, and debuggers, only operate on a per-process or per-machine basis. They do not provide this coherent, end-to-end view of requests, that we need to investigate root causes. Moreover, since each microservice may concurrently process multiple requests, a further challenge is disentangling requests from one another.

Recent work in both academia and industry has begun proposing new tools for monitoring and diagnosing problems in distributed systems. They tackle a range of different important use cases. We give several examples:

Distributed tracing tools such as X-Trace [30], Dapper [1] and Jaeger [2] provide a way to record information while a request executes and tie that information together across different processes and machines after-the-fact. The tools produce *traces* of requests; a trace of a request includes where each part of a request is executed, how long it takes and the events that appear while the execution. Sambasivan et al. [12] gives an overview about use cases distributed tracing tools support including the identification of performance problems, correctness problems and tasks like distributed profiling and request latency monitoring.

Resource attribution tools such as Retro [3] and Stardust [4] provide a way to monitor all of the computational resources that are consumed during a request's execution, on all machines. For example, the CPU cycles consumed, the amount of network traffic, disk usage, and so on. Such information is for example useful to enforce fair resource usage among requests.

Metric-gathering tools such as Census [6, 7] and Pivot Tracing [5] collect and aggregate statistics about the functions being called and dependencies between functions, along the end-to-end paths of requests through the distributed system. Collected metrics can help to automatically generate alerts at an outage or trigger resource up-scaling upon high demand.

Anomaly detection tools such as Pinpoint [31] and Pip [32] track the paths requests take through the system, and watch out for any changes that may occur, such as strong deviations in request latency, changes in the order of work done or incorrect behavior by analysing end-to-end request execution reports.

2.3 Request Flow Tracking

In the above examples, all of the tools have a basic requirement to track work done across multiple processes and machines. The established technique for capturing end-to-end information about requests in distributed systems is *request flow tracking*.

Example: Tracking Requests using Request IDs

To elucidate request flow tracking in more detail, we begin with a simple example of propagating request IDs. When a request arrives at the system, it is assigned a unique request ID. This ID is then conceptually "attached" to the request, and the system includes the request ID anywhere the request goes, such as with remote procedure calls to other services. If the request ID is present everywhere the request goes, then it can be added to any performance and debugging information generated by the system. For example, if there are any interesting or anomalous events that happened, the event will explicitly state the ID of the request that caused it. After the request execution completes, all of the performance and debugging information can be collected from all machines. By grouping this information by request ID, it reveals the end-to-end picture of each request's execution. It also effectively separates information from concurrently executing requests within each process.

2.4 Instrumentation Overview

Context Propagation. The previous example demonstrated how a tool might use a request ID to correlate information generated by different system processes. In general, different tools propagate different identifiers or metadata, but most tools make use of this technique of propagating request-specific metadata. We call this technique *context propagation*.

Instrumentation. Unfortunately, operating systems and language runtimes do not provide context propagation as a primitive that tools can just hook into. Instead, context propagation has to be implemented at the application-level, i.e. directly in the source code of the system. Consequently, tools like distributed tracing often cannot be used out-of-the-box. Instead, some effort must first be spent by the system developers to correctly implement context propagation in their system. Once context propagation is implemented, the tool can hook into it and run correctly. The task of adding context propagation to the system is called *instrumentation*.

Instrumentation is a mostly manual process undertaken by developers. The developer needs to explicitly mark out the flow of a request, in the system source code, for all system components. This cannot be done automatically (e.g. by the operating system) because most non-trivial applications make extensive use of application-level control and data

flow, which are not visible to the operating system. Instead this is a manual task, where developers must exhaustively modify the system's source code to pass around contexts with the flow of requests.

2.4.1 Categories of Instrumentation

When developers modify their systems' source code, the instrumentation required falls into several broad categories:

- **Context Creation:** When a request arrives at the system's request entry point (e.g. an HTTP server or API gateway), a context must be created.
- **Begin Request Execution:** When a request begins execution in a thread, the thread handling the request must store the request's context in a thread-local variable. For as long as the thread handles the request, the request's context can be accessed from the thread-local variable. This usually happens immediately after context creation.
- **Forking:** Often, the thread handling the request might delegate work to other threads, e.g. by enqueuing a work item into a queue that is read by a different thread. When this happens, the request's context must be retrieved from the thread-local variable and saved with the queued work item. This way, the work item remains associated with the request. Forking usually happens for any asynchronous execution patterns such as thread pools.
- **Joining:** When a thread receives work, e.g. by reading from a queue, it also receives the context that was attached to the work item. The thread must store the context in a thread-local variable for as long as it handles work for the request. Joining usually happens for any asynchronous execution patterns such as thread pools.
- **Discarding:** Once a thread finishes work for a request, it must discard the context from the thread-local variable, such that future work the thread processes is not incorrectly attached to the same request. Processed work could possibly be independent of any tracked request. Discarding is for example necessary once a thread completed processing a RPC.
- **Remote Propagation:** When a service makes a remote procedure call (RPC) to another service, it must serialize the request's context and include it with the RPC. The recipient must then deserialize the context upon receipt. The receiver thread stores the context again in a thread-local variable.

2.4.2 Instrumentation API

We can generalize the above categories into an *instrumentation API* that is used by developers when instrumenting their systems' source code. All of the use cases described in §2.2 use some variant of this API.

`create(): void`	creates a new context and attaches it to the current thread (i.e., stores it in a thread-local variable)
`fork(): Context`	retrieves a copy of the current thread's context.
`join(Context): void`	if there is already a context attached to the current thread (i.e., present in the thread-local variable), this operation will combine the provided context with the attached context. If there is no context attached to the current thread, then the provided context will be attached.
`discard(): void`	detaches any context attached to the current thread (i.e., clears the thread-local variable).

In addition, it is assumed that `Context` objects can be serialized and deserialized for propagation over RPC; some tools provide explicit APIs to do this, but we omit them here as they are extraneous to this work.

2.5 Motivating Example

To make these concepts concrete, we present a concrete example of the source-code modifications needed to instrument a *producer-consumer* execution pattern. To provide some additional context, we also illustrate how a distributed tracing tool makes use of contexts to record traces. In our description, we make reference to the instrumentation API of §2.4.2.

Request Flow Overview. Listing 2.1 outlines a snippet of code implementing a producer-consumer execution pattern. We refer to the listing in the following description. We mark in blue the instrumentation needed to propagate context but ignore them momentarily and revisit them after providing an initial overview.

```
1  Queue<Work> workQueue = new Queue<Work>();
2
3  class Work {
4    String name;
5    Context context;
6  }
7
8  // executed by T1
9  startRequest(Request q) {
10   create();
11   compute(q.name);
12 }
13
14 compute(String name){
15   doWork();
16
17   logEvent("delegate work") //E1
18   Work w = new Work();
19   w.name = name;
20   w.context = fork();
21   workQueue.put(w);
22
23   doMoreWork();
24
25   logEvent("waiting") //E2
26   w.await();
27
28   join(w.context);
29   logEvent("work done") //E3
30 });
31
32
33 // executed by T2
34 RunThreadsWith(processingLoop() {
35   while (...) {
36     Work w = workQueue.take();
37     join(w.context);
38
39     logEvent("processing") //E4
40     process(w.name, w); // accessing Work.name
41
42     w.context = fork();
43     w.signalDone();
44     discard();
45   }
46 });
```

Listing 2.1: Demonstrates a common producer-consumer pattern with instrumentation

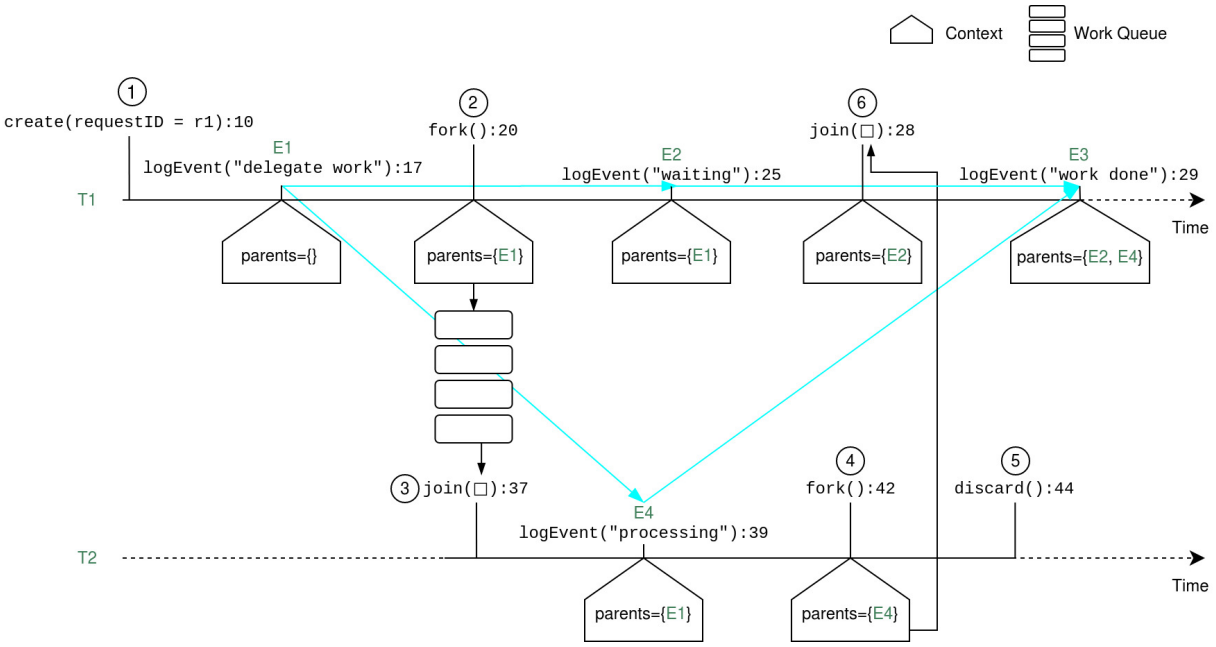

Figure 2.1: An execution of the code from Listing 2.1. Above each thread, we illustrate each line of code invoked (line number attached). Below each thread we illustrate the context at that point of time. We overlay the happened-before relation between `logEvent` calls in cyan.

The request execution proceeds as follows: beginning in thread `T1` (Line 9), the thread does some work, logs an event `E1` with a `logEvent` API, then delegates further work to another thread by enqueueing a work item (Line 21). `T1` continues to execute: it logs an event `E2` then waits for the work item to complete. Meanwhile `T2` receives the work item (Line 36), logs an event `E4`, processes the work item, and lastly signals the work's completion (Line 43). `T1` wakes up and finally logs an event `E3`.

Distributed Tracing. To ground the example in a concrete use case, we now introduce a *distributed tracing* tool into the picture. Before outlining the instrumentation, we will first describe one goal of distributed tracing we focus on for this example.

One goal of distributed tracing is to capture the *causal ordering* of events that appear as part of requests. In this example, under Lamport's happened-before relation $\rightarrow$ [33], distributed tracing would record: `E1` $\rightarrow$ `E2`, `E2` $\rightarrow$ `E3`, `E1` $\rightarrow$ `E4`, and `E4` $\rightarrow$ `E3`.

To achieve this, the tool requires two things. First, to distinguish between events logged by other concurrent requests, the tool generates a unique `requestID` at the beginning of the request. Every `logEvent` call will access and record the `requestID`. Second, to capture happened-before relationships, every `logEvent` will include a reference to the most recent previous `logEvent`; this reference is called the `parentID`. For example, when `E4` is logged, it will include a reference to `E1`. Together, `requestID`s and `parentID`s are sufficient to reconstruct the request trace later from the logged events.

In practical terms, both the `requestID` and `parentID` are provided to `logEvent` calls using context propagation. The variables are stored inside the thread-local context, and accessed whenever `logEvent` is invoked. Figure 2.1 illustrates the happened-before relation with respect to the overall execution; the causal ordering of events is illustrated in cyan.

Instrumentation Needed. We now describe how a developer would instrument Listing 2.1 to track requests. Although we are contextualizing the example for distributed tracing, this instrumentation process is generic and would remain the same for other request-flow tracking tools.

The lines marked in blue in Listing 2.1 illustrate instrumentation for this code.

- `T1` initializes a context using the `create` API function at the beginning of the request (Line 10 in Listing 2.1; ① in Figure 2.1). Internally, this randomly generates a new `requestID`, and stores the created context in a thread-local variable.

- Before `T1` passes the work item to `T2`, it must first invoke `fork` (Line 20 in Listing 2.1; ② in Figure 2.1). This makes a copy of the current context, which `T1` then stores directly with the work item.
- Immediately after `T2` fetches the work item, it puts the work item's context into the thread local variable using `join` (Line 37 in Listing 2.1; ③ in Figure 2.1).
- Once `T2` completes the work item, it invokes `fork` to copy the context from the thread local variable and store it with the work result (Line 42 in Listing 2.1; ④ in Figure 2.1).
- `T2` is now finished with the work item. It invokes `discard` to clear the thread-local variable (Line 44 in Listing 2.1; ⑤ in Figure 2.1). This ensures no other work item begins inadvertently adding `logEvent` to the wrong request in the future.
- Immediately after `T1` receives the result from `T2`, it takes the result's context and invokes `join` (Line 28 in Listing 2.1; ⑥ in Figure 2.1).

Effects of the instrumentation. With contexts being correctly propagated, the distributed tracing tool can use them to pass around `requestID` for the request. Concretely, the `logEvent` API call can internally do the following:

- When any thread logs an event with `logEvent`, the `requestID` can be retrieved from the context and included with the logged event.
- After logging an event, the `parentID` can be updated to the ID of the event that was just logged. For example, after logging `E1`, `parentID` is updated to `E1`.
- When any thread logs an event with `logEvent`, the `parentID` is also retrieved from the context and included with the logged event. For example, when `E4` is logged, the `parentID=E1` will be logged.

The end result is a correct trace: every event records its immediate predecessor events according to the happened-before relationship $\rightarrow$, and every event records the request ID that it belongs to.

Although we provided distributed tracing as a concrete use case in this example, the instrumentation API calls marked in blue throughout are generic. Except for small adjustments any tool that uses request flow tracking would need to insert instrumentation as presented for this example.

Pathfinder With respect to the example, Pathfinder specifically targets the *instrumentation* API calls, marked in blue throughout the example. This instrumentation is a pre-requisite for deploying *any* tool that uses request-flow tracking. For example, if a developer wished to deploy distributed tracing, they would first have to make all of the instrumentation source code modifications.

Pathfinder's goal is to aid the developer in determining where to insert these instrumentation API calls. In the next chapter we will motivate how instrumentation is a challenging task for developers today, due to system source code complexity and a lack of useful tooling.

2.6 Execution Patterns

The example in §2.5 illustrates one example of an *execution pattern*. Execution patterns typically have *multiple, interrelated instrumentation points*. Successfully instrumenting the producer-consumer pattern in the example required more than one modification. If any of these modifications were omitted or incorrectly placed, the instrumentation would not correctly capture the flow of execution.

Conceptually, instrumenting a system happens at the level of execution patterns, rather than just individual code locations. It involves finding the execution patterns used by the system, then correctly identifying the multiple relevant instrumentation points.

Beyond the producer-consumer example given, there are a range of different execution patterns that can be instrumented. They usually involve taking a context from a thread, and ultimately, passing it to a different thread, where execution continues. To aid in the reader's intuition of instrumentation, we provide an overview of several execution patterns commonly found in systems.

Intra-Process Execution Patterns. Similar to the producer-consumer patterns, these are internal to a process and influence how a request execution proceeds.

- **Thread Fork.** The current thread T1 starts another thread T2 to continue the execution of the request. T1 may terminate, continue with the execution of the same or another request, or wait for T2 to finish.
  Instrumentation: In this scenario the context of the current thread is propagated to T2 when it is started. If T2 joins with T1 when T2 completes, the context can be propagated back to T1.

- **Thread Messaging.** The current thread stores a message in a queue or similar data structure. Then a different thread fetches the message from the data structure to continue the execution. In rare cases the data structure might be replaced by single shared variables. This pattern is extremely common and appears in execution services that use a thread pool to handle arbitrary work, in event systems where the message describes an event that is triggered asynchronously, or in custom execution infrastructures that realize application specific batching, interleaving or scheduling strategies.
  Instrumentation: A possible instrumentation was already presented as part of §2.5.
- **Request Interleaving.** The current thread stores an intermediary state of a request or work object R in a data structure to continue with the execution of another request or work object to later resume the execution of R. If several threads are involved in processing the different work or request objects the pattern becomes similar to the previous one.
  Instrumentation: On pause events the context is retrieved from the thread and attached to R. The thread discards the context and on resume the context is again picked up from the work item by the thread before the execution is continued.

Inter-Process Execution Patterns. End-to-end requests span multiple machines, and instrumentation is required for including contexts with remote communications.

- **Messaging.** The current thread sends a message to a thread in another process (e.g. via RPC). Depending on if the processes are on the same machine the message is transferred over network, local sockets, pipes or shared memory.
  Instrumentation: In this scenario the context is retrieved from the current thread with `fork`, serialized, and included in the message header. On the receiver side the other thread retrieves the context from the message, deserializes it and calls `join`.

2.6.1 Summary

While the above list outlines commonly-used execution patterns, it is not exhaustive. Execution patterns are an application-level design choice; the original system developers could have implemented arbitrary custom patterns for the logic of their application. We must be aware of this when instrumenting a system, since all executions patterns must be instrumented.

Although instrumentation must be done with care, the resulting source code modifications are usually modest in size, ranging only into the tens or hundreds of lines of code (LOC) modification [3, 5]. For example, in our evaluation, instrumenting Cassandra and HBase required only 19 respectively 36 LOC modification, compared to a source code base of 385252 and 406263 LOC.

Nonetheless it is a significant challenge to identify and instrument execution patterns, and Pathfinder's goal is to aid developers in this task. In the next chapter we will examine this challenge in more detail.

Chapter 3

Challenges

Although instrumentation is conceptually straightforward, it is challenging in practice. Developers must exhaustively identify all relevant execution patterns, then correctly and completely insert the appropriate instrumentation API calls. In this chapter, we examine the reasons why instrumentation is a challenging task for developers.

3.1 How do Developers Approach Instrumentation?

By default, developers must identify and instrument the aforementioned execution patterns themselves. Since the goal is to capture the complete execution path of requests, instrumenting a system means identifying and instrumenting all instances of execution patterns that are used by the system. In this section, we discuss the typical approaches developers take in completing this daunting task.

3.1.1 Recommended Approaches in Developer Guides

Several high level instrumentation guides provide recommendations for how to build up system instrumentation [34, 35]. The guides recommend the following steps:

1. Specify the set of high value requests that should be tracked. Start by instrumenting the most important ones.
2. Identify specific libraries or services that are the most widely used (e.g., RPC libraries, web frameworks) . Instrumenting these libraries will give the broadest coverage.
3. Add instrumentation on intra-process level as needed.

The first step of this guide depends on the system and use case and can also include important operations triggered by system internals. Internally, systems usually re-use common execution structures such as thread-pools, and there may not be a 1-to-1 mapping between request types and the internal structures that must be instrumented. This is reflected in the second recommended step: instrumentation often boils down to instrumenting some 'core' execution libraries and frameworks that are used throughout the whole system. We found this to be true in our instrumentation of Cassandra [26] and HBase [25] used in our evaluation in Chapter 7, and from our prior experience instrumenting other systems [5, 3, 14].

The third recommended step encompasses the vast range of instrumentation needed in practice, both in terms of code modifications and developer effort. To guarantee that context flow is end-to-end, request context must be propagated form one side of the process to the other, so that context flows along the critical path to connect incoming RPCs with outgoing RPCs. If context is not propagated at an intermediary component on this critical path, the context flow cuts of and subsequent components cannot be associated with the current request (illustrated with Figure 1.1 in Chapter 1). Context propagation to components outside of this critical path is optional and depends on the target granularity and higher-level task that finally utilizes the context propagation infrastructure. An example of such an optional case is context propagation to sub-tasks that do not need to complete for the completion of the whole request.

3.1.2 Code Traversal

Unless developers have a perfect mental model of the system's entire code base, they will need to do some amount of exploration to identify the paths requests take through the system and the execution patterns they touch. This

manual traversal can take several forms. This can happen systematically along request paths, with a global search for keywords and interfaces, or through trial and error using partial instrumentation.

Traversing the code base along a request starts where the request is received and follows the request flow through the code base until the request reaches a process boundary or ends. Along this path context propagation points are identified and instrumented. This procedure helps to derive a global execution model of the system and puts instrumentation in a context what makes it easier to asses the affects and makes is less likely to miss context propagation points. We consider this approach as adequate in the context of intra-process instrumentation where there is initially little indication of where instrumentation is needed.

There are also other ways to find instrumentation points without any request context. One is looking for important infrastructures like global execution services or inter-thread communication components. This approach relies on searching for keywords or interfaces that appear around RPC libraries etc. We consider this approach useful in contexts where the instrumentation effects to the system are obvious and completeness is guaranteed, as when instrumenting a global inter-process communication infrastructure. However, this procedure is barely helpful for intra-process instrumentation. The intra-process context exhibits various inter-thread communication patterns which cannot be revealed with a simple search for keywords or interface implementations, leading to an incomplete instrumentation. In addition, the full effect of one instrumentation point is often hard to asses without the context of a request.

In general, code traversal is very ad-hoc. Developers have no indication that their instrumentation is either complete or correct, and there are few hints about how to progress in instrumenting a system.

Although manual code traversal is a somewhat “brute force” approach to instrumentation, it is nonetheless the de facto approach to instrumenting systems. Later in this chapter we describe some attempts to automate the instrumentation process, and the corresponding problems that arise. We know of no prior work in tools that aid developers in doing instrumentation.

3.2 Instrumentation Pitfalls

Instrumenting systems is a tedious and time consuming task. Researchers and practitioners describe it as one of the most time consuming and difficult parts of deploying a tracing tool [11, 12, 13, 36]. Especially the instrumentation in the intra-process context is consistently described as difficult:

Facebook report about instrumentation of android apps [37]:

> *"What’s worse, instrumenting the code thoroughly was difficult due to the multi-threaded nature of the app and the highly asynchronous nature of user interactions. Every time the app submitted work to another thread, a little identifier would have to be handed over to stitch things together. Canceling markers and making sure things were connected correctly made the experience less than ideal."*

Nike report about deploying distributed tracing [22]:

> *"In reactive non-blocking frameworks and libraries, this thread-hopping is the norm, not the exception. This means that supporting [intra-process propagation] in Java can get ugly, depending on what you’re doing."*

As mentioned in §2.3, one way to point out places where context propagation is needed is via manually traversing the code base along the flow of a request. This task is comparable to manually doing a static code analysis that includes heuristically inferring likely runtime types and values, following many many branches deep down into dependencies and understanding what is actually called when code is indirectly integrated via reflection, dependency injection or if dynamically loaded. It is easy to get lost in this difficult search scenarios.

Developers miss instrumentation places, implement them incorrectly or forget to adjust them after the application logic was changed.

3.2.1 Pitfall: Incorrect Instrumentation

If request flow is not understood at execution boundaries, incorrect instrumentation leads to wrongly attributing work to requests. As part of instrumenting a thread-pool in Accumulo [38], context was propagated to threads when they are initially created instead of when they start to process a work item as part of a request flow [16]. This lead to mistakenly associating the entire activity of a thread with the request that initially started it. Similarly, if developers fail to mark the end of a request’s execution, the next request will inherit the previous request context. For example, Accumulo’s instrumentation did not mark the end of a compaction task once it completed, causing the next compaction task to

be merged into the same trace [39]. In contrast to this, prematurely discarding of the request context means that subsequent operations stay hidden and are not linked to any request. For example in HBase, RPC call tracing was inadvertently terminated before the call actually completed [40], leading to wrong request durations.

3.2.2 Pitfall: Incomplete Instrumentation

Inexhaustive instrumentation is a common pitfall, particularly for infrequently exercised code paths such as background tasks, edge-case handling, and failure recovery. For example, Cassandra [26] did not initially instrument request retries [41], so traces would end after the first request attempt; nor did Cassandra instrument result pagination [42]], so traces would end after the first page of results. HBase produced traces with gaps in them due to its uninstrumented write-ahead log [43]. Instrumentation omissions in Cassandra failed to propagate request contexts to the 'finalize' step of queries, leading users to incorrectly believe requests were not completing properly [18, 44]. An uninstrumented thread pool caused request flow tracking in HBase' tracing infrastrucutrue to end prematurely without tracking any server-side activity [19]. Similarly, missing context propagation to thread pools in Accumulo [17] lead to erroneous loss of context on some request paths.

3.2.3 Pitfall: Changes Break Instrumentation

After extending, refactoring or adjusting application code, context propagation must also be adjusted if necessary. But this can easily be forgotten. For example, a patch to Hadoop's inter-process communication [45] forgot to include HTrace context in the new call headers [46]. A patch to refactor components in Cassandra [47]. introduced some new execution boundaries, but failed to propagate request contexts across these new boundaries [48, 49]. Similarly, refactoring broke tracing support for several file system operations in Accumulo [50].

3.2.4 Summary

These pitfalls demonstrate the importance that instrumentation be *complete* and *correct*. They also illustrate how maintaining instrumentation in the long term is important. Taken together, these pitfalls motivate a need for tools that make it easier and faster to implement correct and complete instrumentation.

3.3 Can we Automate Instrumentation?

Facing those challenges rises the question: *Can we further automate or eliminate the instrumentation process?* Intuitively, instrumentation revolves around identifying and modifying execution patterns, as we described in Chapter 2. Surely, then, this task might be amenable to some amount of automation, since there are common execution patterns and therefore, presumably, common solutions for instrumenting them. Unfortunately, we believe fully automated instrumentation to be beyond the realm of possibility due to the *subjectivity* of instrumentation.

3.3.1 Pitfall: Subjective Request Paths

A request takes a specific path through a distributed system. As we showed, request context must follow this path such that request related information can be associated with the request using the context. However, it is subjective what this path is and so where the context should go. We give an example to illustrate this aspect.

Figure 3.1 shows the execution of two requests. As part of its execution, *Request 1* stores an object in a cache, finishes and returns to the client. After a period of time the execution of *Request 2* starts. *Request 2* accesses the cache and sets a new object into the cache. This causes the old object to be evicted. The eviction procedure includes a write out of the old object to disk that is performed as part of *Request 2* before it returns.

In this example it is possible to associate the eviction procedure with *Request 2* that finally performs the eviction or with *Request 1* which originally caused the work. Sambasivan et al. [12] introduced this example and calls the slice that connects the eviction procedure with *Request 1* the *submitter-preserving* slice and the slice that connects the eviction procedure with *Request 2* the *trigger-preserving* slice. So the two slices shown in Figure Figure 3.1 show the same executions of the two requests only the attribution of work differs. Corresponding to which relation should be preserved the propagation path of the context differs. Propagating the request ID of *Request 1* to the eviction procedure (executed by *Request 2*) allows to associate the caused disk utilization with *Request 1*. This might be useful in the context of resource attribution tools. On the other hand, propagating the request ID of *Request 2* to the eviction procedure, associates the task with *Request 2* what might be useful for latency diagnosis. If the eviction is part of the critical path of requests similar to *Request 2*, this might help do understand the high latency of a request. If we are

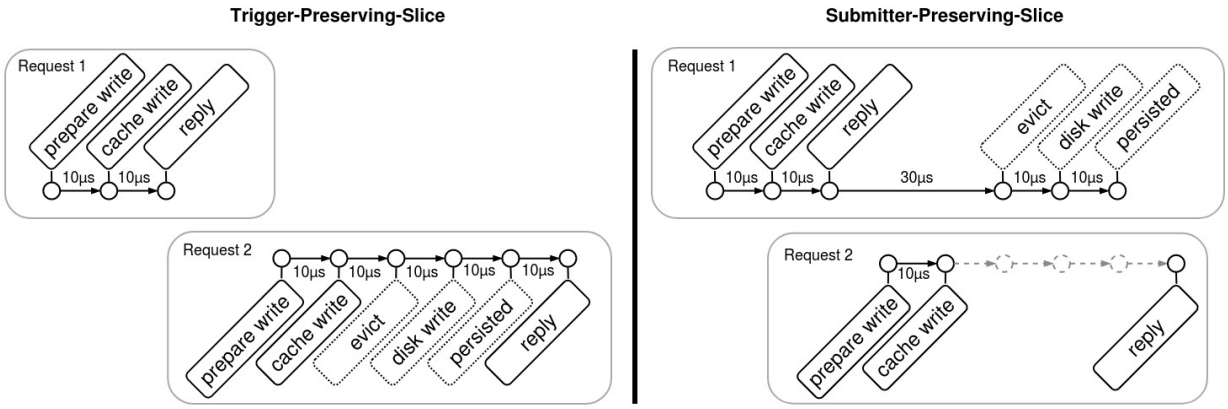

Figure 3.1: Trigger-preserving and submitter-preserving slice of request executions

only interested in one of the use cases, it does not make sense to capture both relations, so to associate the eviction procedure with both requests. Unnecessary information are not useful and it would be needed to filter them out later when investigating the data.

3.3.2 Pitfall: Subjective Granularity

Even for simple and obvious request paths it is often not desirable to capture all relations on the lowest level of granularity including all relations to all execution parts. We show another example to illustrate this aspect.

When a distributed tracing tool is deployed to a storage system to be used by a system end-user to monitor system performance to spot slow nodes or data hot spots, it does not make sense to include information on the traces that point to system internals the end-user cannot influence. In this case it is not necessary to propagate the context further down to system internals. However, if a distributed tracing tool is deployed to rather be used by the system developers for distributed profiling to improve the system design, capturing the fine-grained execution graph can make sense. In this case propagating the context even to minor tasks can be desirable.

In general, which relationships should be preserved depends on the tool that uses the request tracking capability, the system and the use case. Based on these factors developers need to consciously decide in the individual case what information in the system should be associated with which request. Those decisions are ultimately reflected in the context propagation infrastructure, which must be implemented accordingly. Preserving just all causal relationships introduces unnecessary overhead, development costs, and only moves the filtering problem to a later point.

3.3.3 Pitfall: Unclear Instrumentation

Another aspect that hinders automation is that there exist execution patterns for which there is no general agreement on how to instrument them [51]. For example when several request executions join at some point and only one operation is executed for all of the requests, there are several ways to associate this one operation with the requests. Usually the instrumentation here has to be aligned with the capabilities and use case of the target request flow tracking tool.

3.3.4 Summary

These pitfalls motivate why fully automatic instrumentation approaches are infeasible in practice. Instrumentation is qualitative and use case dependent, making it intractable to automate completely and correctly. Even if the approaches *can* arrive on a correct solution, or several candidate solutions, it would still require developer input to validate the correctness of the instrumentation. Regardless, no fully automated instrumentation approaches exist today.

3.4 Can Partial Automation Help?

Although fully-automated instrumentation is a panacea, there are several approaches that partially automate some aspects of instrumentation, with a goal of relieving the developer instrumentation burden. We provide an overview of these approaches here, followed by some of the associated pitfalls.

3.4.1 Approach: Pre-Instrumented Libraries

Projects like OpenTelemetry [7] and OpenTracing [52] try to reach a standard for context propagation to provide vendor-neutral APIs. With a standard it is possible to share instrumentation of services, common libraries and execution patterns of various runtimes and programming languages. This allows, for example, to use an existing instrumented version of a common RPC library that internally propagates the context along RPCs. Since context propagation is already realized inside the library, placing propagation code manually is not longer necessary at this point.

Naturally, a developer did instrument the source code at some point. The main benefit of pre-instrumented libraries is that instrumentation can be *reused*. This is especially effective if a system's main execution patterns reside within the library. For example, the library might provide an executor or futures pattern, along with all of the required instrumentation.

While this can reduce the amount of additional instrumentation the system needs, it comes with its own set of associated problems:

- Distributed systems tend to be heterogeneous [12], so use a brought set of different libraries and runtimes. So there might not exist a pre-instrumented version of a used library or service.
- Context propagation is use case dependent [12]. Depending on the top-level task that finally uses the context propagation infrastructure, existing pre-instrumented libraries might not provide the desired instrumentation.
- Especially in the context of multi-threaded execution, there exist many custom execution patterns which only use common data-structures for which a general instrumentation does not make sense and so does not exist.

3.4.2 Approach: Exploiting System Visible Channels

Approaches like Causeway [23] rely on events that are visible to the operating system, such as using pipes and other process communication mechanisms. When the operating system has visibility of these, it can automatically propagate request contexts across those channels for send and receive events. Recent work has also included capturing signal/wait events and thread forks [10], and network send and receive events [24].

While such an approach is appealing, it has a clear limitation: most execution patterns are implemented at the *application level*, and the operating system lacks visibility into communication events. Typically, the application is a black box, that receives multiple requests as inputs, and makes multiple subsequent calls to other services. Tying up inputs to outputs is a challenging problem, and instrumentation is needed to explicitly address this.

3.4.3 Approach: Exploiting Coding Paradigms

Other approaches rely on specific coding patterns or frameworks. That is, they expect all execution patterns used by the system to be encapsulated within a specific library, framework, runtime, or programming construct. Outside of this "golden path", instrumentation will not be automatic.

AppInsight [8] and Timecard [53] realize automatic context propagation along user transactions through mobile apps. Thread execution boundaries within processes are identified with a simple heuristic. In situations like asynchronous event execution, callback handling and concurrent processing, the event logic, the callbacks or the work items must be transferred across execution boundaries exclusively in the form of specific functional objects having a specific run-time type. If this condition is not full filled corresponding execution boundaries are not instrumented. Even if this might be sufficient under a specific framework in the domain of mobile apps, the approach cannot be generalized and leads to incomplete and/or overcomplete context propagation in other systems. Similarly, WebPerf [54] requires that a specific coding paradigm for asynchronous execution is used. Outside of this framework the approach is not effective.

3.4.4 Approach: Pattern Matching

Retro [3] and Pivot Tracing [5] automate the instrumentation of common patterns, by using AspectJ to identify uses of Java interfaces such as `Thread`, `Runnable`, `Callable` and so on. The authors noted that this approach reduced, but did not remove, the need for manual instrumentation. In several cases, pattern matching inserted instrumentation *incorrectly* in places where it was not needed; the authors had to manually disable those instrumentation places. Similarly, Domino [55] provides automatic rewriting of JavaScript applications through a limited set of interfaces.

The limitations of such a pattern matching approach are clear: it is best effort, and will not capture anything outside of a narrow "golden path". Moreover, it can (and did) incorrectly add instrumentation where it was not required. The

end result is that a developer must still manually validate the correctness of the instrumentation, and ultimately fix any errors directly in source code.

3.4.5 Approach: Analyzing Inter-Thread Data Flow

Whodunit [9] provides a more general approach that tries to automatically propagate context along relevant data flows of a transaction within a process. To realize this, Whodunit analyses producer-consumer relations of threads through shared memory to infer transaction flow. This approach is in principle applicable to arbitrary systems. However, to identify relevant data flows, Whodunit relies on several assumptions that are inappropriate for many distributed systems.

- "Threads in a multi-tier application have predefined roles. They are either producers or consumers of a resource, but not both producers and consumers of the same resource.." – Whodunit ignores dataflows that violate this rule and miss context propagation in that case. In the context of execution services and thread pools it is common practice that a worker thread submits a work item as part of processing another, which is later processed by the same thread; or in situations where a request is paused and interleaved, a request object might be re-enqueued and picked up again at a later point in time by the same thread. In those cases the same thread acts as producer and consumer but still context propagation is needed.
- "Accesses to shared data structures always occur in critical sections protected by locks" – especially in concurrent data structures the use of locks is often avoided for performance reasons. In such cases context is not propagated where it should be.

Additionally, the presented techniques are essentially restricted to a specific use case. Whodunit presents a dynamic approach that decides at runtime where context should be propagated. The presented techniques cannot be easily adapted to give the user the ability to control where context should be propagated. But as we discussed in §2.3, different use cases require different context propagation infrastructures.

3.4.6 Pitfall: Need for Developer Intervention

The approaches listed in this section all share a common pitfall: a developer must ultimately intervene to validate the correctness of the instrumentation, fix any issues with incorrect instrumentation, and manually add any instrumentation that was missing.

All the shown approaches suffer this pitfall. The instrumentation based on pre-instrumented libraries, code patterns or coding paradigms might cover an arbitrary non-connected subset of all existing request execution boundaries. Those approaches do not give the developer any information about if instrumentation might be missing, where to extend the instrumentation or if existing instrumentation might be irrelevant.

Arguably, developers must pay a *higher* cognitive cost for intervening at the source-code level under these approaches. Not only does it require identifying and understanding the execution patterns in the system; it also requires identifying which of these get automatically matched and instrumented, versus which don't. No existing approaches offer tools or solutions to aid in this task; fixing instrumentation happens by manually traversing source code.

3.4.7 Summary

So overall we can summarize that context propagation is required by many request flow tracking tools but does not come for free. Developers have to manually modify code at various potentially non-obvious places throughout a system's code base what is time consuming and difficult. The key challenges arise from a need for intra-process context propagation, which are constrained by large source code bases and the infeasibility of automation.

In the remaining chapters, we will present the design, implementation, and evaluation of Pathfinder, a tool for instrumentation that takes a different perspective. Unlike prior work, Pathfinder is designed with the expectation that developers *are* involved in the instrumentation process. The foundational challenge remains the same: making it easier to identify execution patterns in large source code bases. Its goal, however, is to speed up instrumentation, with developers making all choices about where and how instrumentation should be placed.

Chapter 4

Design

In this chapter, we provide an overview of Pathfinder's design. We start by summarizing our goals, then present our key intuitions, followed by the design of Pathfinder.

4.1 Goals

Pathfinder is a **developer-in-the-loop tool** that assists the **manual instrumentation** of distributed systems for **intra-process context propagation**. As we outlined in Chapter 3, a key challenge for instrumenting large systems is that developers must manually traverse the code base to perform instrumentation – a significant challenge since many real-world distributed systems can be very large. Concretely, for example, the two systems we examine in our evaluation, Cassandra [26] and HBase [25], comprise 385252 and 406263 lines of code respectively, spanning 4621 and 3427 classes, and 1678 and 1919 files. Instrumentation, by contrast, is only relevant to a small fraction of the code where execution patterns are implemented. Pathfinder's core objective is therefore to narrow developers' focus to only the locations in code where instrumentation is relevant.

To achieve this core objective, Pathfinder must satisfy several goals related to quality of instrumentation and developer effort.

G1: Completeness. Request flow tracking is only effective when the instrumentation is complete, as outlined in §3.2. Pathfinder should exhaustively identify all possible intra-process instrumentation points such that the final instrumentation covers full request paths.

G2: Correctness. Incorrect instrumentation leads to incorrect request paths. Pathfinder should make it as easy as possible for developers to make correct instrumentation choices, by reducing the amount of information presented to developers and by subdividing the problem into manageable pieces.

G3: Fast. Instrumenting a system is time-consuming, and the effort required is open-ended. Instrumenting a system using Pathfinder should be both fast and quantifiable in terms of time.

G4: Incremental. Full request paths are a requirement for completeness, but as we discussed in §3.1.1, in the most cases not all request *types* need to be exhaustively instrumented.

G5: Composable. Pathfinder should be sensitive to any existing instrumentation that may exist in the system.

G6: Developer First. All instrumentation decisions should ultimately be made by the developer. Pathfinder's focus is on providing a narrow set of possible instrumentation choices, to make it easier for developers to make this choice.

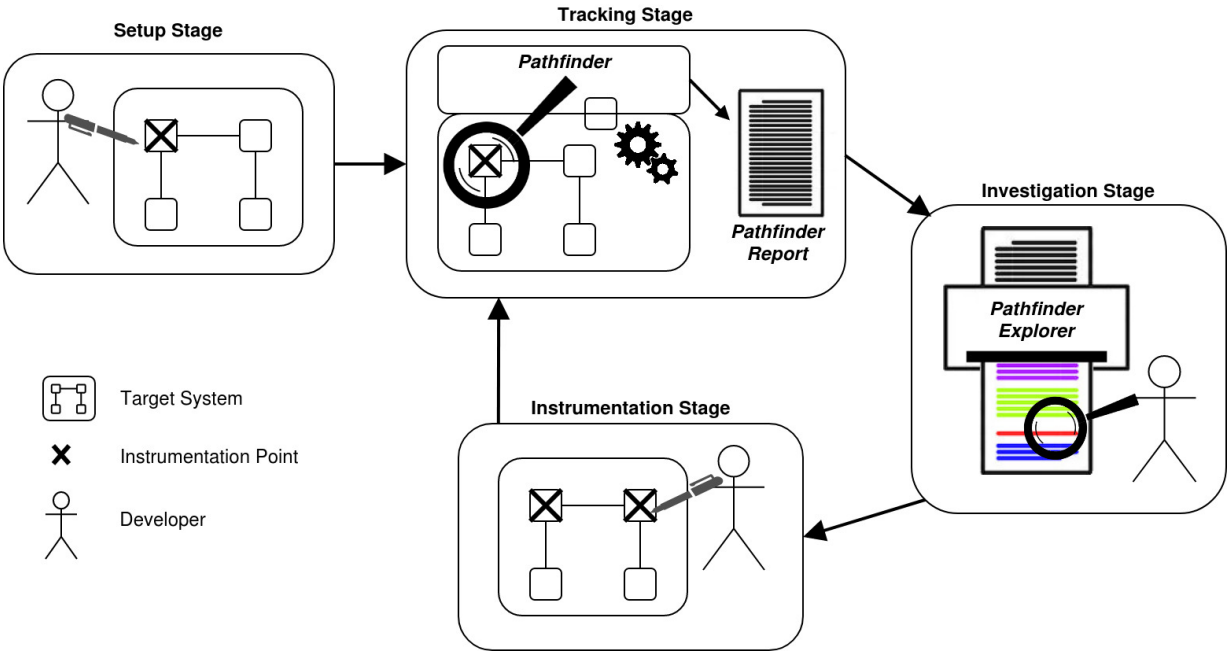

Figure 4.1: An overview of how developers use Pathfinder to instrument a system.

4.2 Pathfinder Overview

We begin with an overview of how a developer uses Pathfinder to instrument a system. Our goal is to provide an overview of Pathfinder's main components; in the subsequent sections we provide more details of each component.

A developer uses Pathfinder during development time to identify candidate instrumentation points. Pathfinder is not used thereafter; once the instrumentation process is complete, Pathfinder's job is done and the developer is free to use the detected instrumentation points with any other request-flow tracking tool.

Using Pathfinder is an iterative process. After an initial setup stage, the developer proceeds in iterations. Each iteration involves a tracking stage, an investigation stage, and an instrumentation stage. The developer repeats iterations until instrumentation is complete. Figure 4.1 illustrates this process.

Setup stage. To start using Pathfinder, the developer must select a *starting point* in code. The starting point can be arbitrary, but the intention is to begin at the logical starting point of a request. On the client side where service API calls are initially triggered or at the RPC entry points where API calls are started to be processed. The developer inserts instrumentation at this one starting point.

Tracking Stage. The developer then runs the system with Pathfinder attached. Pathfinder performs runtime analysis of the program to detect possible locations where new instrumentation could be inserted. Thus the developer must submit a workload to the system or execute test cases to trigger different code paths in the system. The workload does not need to be exhaustive – it only needs to execute the code paths the developer wants to be instrumented. The profiling stage is brief – on the order of minutes – before Pathfinder has collected sufficient output for the next stage. This output is called a *Pathfinder Report*.

Investigation Stage. Developers then use a tool we have developed called *Pathfinder Explorer* to consume the report from the tracking stage and deduce where instrumentation should be inserted. The report includes numerous possible locations where new instrumentation could be inserted, and details class names, method dependencies, and variable names that are likely to be part of some execution pattern. The developer interprets this output, possibly cross-referencing with source code, to decide any places where instrumentation might be necessary.

Instrumentation Stage. Finally, the developer inserts one or more lines of instrumentation in the system's source code. Pathfinder exposes the same instrumentation API as the one outlined in §2.4.2. Once new instrumentation is inserted, the developer repeats the loop starting from the tracking stage. During the next iteration, the developer will

see different output as a result of the new instrumentation, which helps Pathfinder guide where its runtime analysis should apply.

Overall, using Pathfinder instrumentation is no longer a manual code traversal task. Instead, Pathfinder guides the developer by quickly identifying points where the developer should focus their attention. Pathfinder collects enough contextual information to enable developers to understand the relationship between points in code that comprise some higher-level execution pattern.

4.3 Inter-Thread Communication (ITC)

A key technical challenge for Pathfinder is how to detect possible instrumentation points. Our intuition to address this challenge is that *instrumentation aligns with inter-thread communication*. We reach this intuition by examining the instrumentation of a number of large open-source distributed systems, including HBase [25], Cassandra [26], HDFS [56], Hadoop MapReduce [57], YARN [58], SockShop [59], DeathStarBench [60], ZooKeeper [61], and Spark [62].

Concretely, instrumentation is only required at request execution boundaries: places where a request begins or ends executing in a thread. A thread cannot know what to execute without receiving some information from elsewhere, such as a work item, and similarly if it does not directly write its response to the network, then it must pass the result to some other thread. Even if a thread re-enqueues a work item into its own request queue, there still exist *some* detectable inter-thread entry and exit points[1].

Inter-thread communication is a key building block for Pathfinder. Inter-thread communication is intrinsically a read-after-write relation. Information between threads are exchanged through shared memory where one thread writes a value another one reads. The goal of Pathfinder's runtime analysis is to identify memory addresses with an inter-thread read-after-write relation. Most memory reads and writes do not belong to such inter-thread read-after-write events, so restricting developer attention to these locations immediately reduces the search space in comparison to a manual code search.

Definition 4.3.1 (Inter-Thread Communication Event)**.** An inter-thread communication event between thread A and thread B occurs at time t_2 if the following conditions hold:

- Thread A writes to a memory address X at time $t1$.
- Thread B reads from memory address X at time $t2$, with $t1 < t2$ and $A \neq B$.
- There does not exist a point in time $t3$ where $t1 < t3 < t2$ and a write happens to memory address X from any thread.

Thread A is considered the producer/writer and thread B is considered the consumer/reader. There can be multiple ITCEs for one write.

4.3.1 Example

We provide an example of this observation in Listing 4.1 which shows a slightly simplified version of the code illustrated earlier in Listing 2.1. As before, we highlight context propagation instrumentation in blue. In the example, inter-thread communication occurs for the following fields:

- `Work.name` – set by `T1` on Line 18 and read by `T2` on processing the work item at Line 34.
- `Queue.first` – pointing to the head of the queue. `T1` writes to this field when submitting the work to the queue at Line 20 and `T2` reads from it when dequeuing at Line 31.
- `Queue$Node.item` – internal to the implementation of `Queue` is a class `Node` with a field `item`. Each entry in the queue is stored as an `item` on a `Node` instance. `T1` writes to this field when submitting the work to the queue at Line 20 and `T2` reads from it when dequeuing at Line 31.

Though this is a simplified example, in real system code methods such as `compute` and `processingLoop` may be defined in disparate locations, making manually finding these methods and their relationship to one another challenging. By contrast, Pathfinder's tracking stage would identify inter-thread communication events for the memory addresses of these variables.

[1] We discuss the pathological case of a single-threaded application in Chapter 8

```
1  Queue<Work> workQueue = new Queue<Work>();
2
3  class Work {
4    String name;
5    Context context;
6  }
7
8  // executed by T1
9  startRequest(Request q) {
10   create();
11   compute(q.name);
12 }
13
14 compute(String name){
15   doWork();
16
17   Work w = new Work();
18   w.name = name;
19   w.context = fork();
20   workQueue.put(w);
21
22   doMoreWork();
23
24   w.await();
25 });
26
27
28 RunThreadsWith(processingLoop() {
29   // executed by T2 and others
30   while (...) {
31     Work w = workQueue.take();
32     join(w.context);
33
34     process(w.name, w); // accessing Work.name
35
36     w.signalDone();
37     discard();
38   }
39 });
```

Listing 4.1: Demonstrates a common producer-consumer pattern with instrumentation

4.3.2 Naïve ITCE Tracking

A simple approach to detecting ITCEs in a program execution is as follows: When a thread writes to any memory address we associate its thread ID with the address in a global mapping. When afterwards a thread with a different thread ID reads from the same memory address we detect an ITCE between the writer and reader thread.

There are two problems with this approach:

- **Overheads.** Such an approach would invoke Pathfinder when reading and writing every single address in the whole address space. This would impose impractical overheads.
- **Context.** Pathfinder needs to establish a connection between memory accesses and code locations. This includes where in the code the memory addresses are accessed and to which classes, object fields or arrays they belong.

4.3.3 Efficient ITCE Tracking

So Pathfinder takes a similar approach, but only tracks a small subset of the overall address space. We observe that it is only possible for memory to be used in an ITCE when it corresponds to one of the following programming language constructs:

- global class fields
- static class fields
- arrays (allocated space)

Concretely, for common programming languages the shared memory address space in which ITCEs can happen is a subset of the address space reachable through these language constructs. In practice, it is substantially smaller than the full address space. Pathfinder only needs to track this smaller subset of the address space, without losing detection accuracy. To achieve efficient ITCE tracking, Pathfinder analyzes the program's source to identify the subset of fields and arrays that must be tracked. We refer to these as *tracked memory addresses*. Pathfinder ignores local fields that can never have ITCEs.

4.4 Communication Slices

Detecting ITCEs alone is not useful for developers, because an ITCE only comprises a memory address, which alone is not useful to developers. Our second intuition is that developers require higher-level, abstract information that maps memory addresses back to code locations where instrumentation might be needed. Instrumentation is done at the source code level, and conceptually aligns with execution patterns. Many ITCEs might map to the same locations in code, and a single execution pattern can comprise multiple source code locations.

We refer back to the example in Listing 4.1 to illustrate. Two of the three ITCE detections occur within the `Queue` class. Even if we could immediately map memory locations back to specific fields in the program's source, it would not be illuminating to developers because `Queue` is widely used. It would be sub-optimal to insert instrumentation directly in the `Queue` class, because it is used in many other places, where context propagation is usually *not* necessary. In the example, a more appropriate instrumentation location is `compute` and `processingLoop` as marked in blue.

This is obvious to a developer – though not to an automated tool – and only when the developer is given sufficient information about where the calls to `Queue` came from. To make this instrumentation decision, the developer would need to inspect the *stack traces* of both the writer and reader thread at the point when the ITCE occurred. The stack traces detail more information about the specific paths taken by the program to read and write those fields. From this information, the developer can find the place in code where the *execution pattern* occurred. Locating the execution pattern usually entails looking several hops back up the stack.

Definition 4.4.1 (Communication Slice)**.** A communication slice corresponds to a inter-thread communication event (Definition 4.3.1) and includes:

- Thread IDs of the ITCE's writer and reader thread.
- Stack traces of the ITCE's writer and reader thread.
- A *memory identifier* that corresponds to a location in code (e.g. field name). We elaborate memory identifiers in §4.4.2

Field	Writer Stack	Reader Stack
`Queue.first`	`Queue.put(Queue.java:93)`	`Queue.take(Queue.java:68)`
	`Example.compute(Example.java:20)`	`Example.processingLoop(Example.java:31)`
	`Example.startRequest(Example.java:11)`	`Thread.run(Thread.java:742)`
`Queue$Node.item`	`Queue$Node.<init>(Queue.java:13)`	`Queue.take(Queue.java:68)`
	`Queue.put(Queue.java:93)`	`Example.processingLoop(Example.java:31)`
	`Example.compute(Example.java:20)`	`Thread.run(Thread.java:742)`
	`Example.startRequest(Example.java:11)`	
`Work.name`	`Example.compute(Example.java:18)`	`Example.processingLoop(Example.java:34)`
	`Example.startRequest(Example.java:11)`	`Thread.run(Thread.java:742)`

Table 4.1: Inter-thread communication slices when executing Listing 4.1. A slice comprises the *Field* that has an inter-thread read-after-write relationship; a *Writer Stack* showing where the field is written; and a *Reader Stack* showing where the field is read. The writer stacks were truncated to the relevant part. We list the above fields in *detection order*; that is, when the program runs, it first reads `Queue.first`, then `Queue$Node.item`, then `Work.name`.

4.4.1 Example

Table 4.1 outlines the three slices in this example. It indicates that several fields are read and written in close proximity to one another, in `Example.compute` and `Example.processingLoop`. Method names are often descriptive, providing additional clues about the existence here of some execution pattern. The following can be derived from these slices:

- As part of executing `compute`, the writer thread adds an item to a queue. This item is dequeued from the queue by the reader thread as part of executing the method `processingLoop`.
- A `Work` object is created immediately preceding `Queue.put` in `compute`. This object is accessed by the reader thread right after the execution of `Queue.take` in `process`. This gives a strong indication that the writer thread delegates some work to the reader thread that processes it.
- The methods `compute, process, processingLoop` trigger the communication. The methods `Queue.put/take` belong to a generic data structure that is not intrinsically related to any specific use case.

4.4.2 Capturing Communication Slices

To capture communication slices, we extend the ITCE detection mechanism that we described in §4.3.3. Firstly, when a thread writes to a tracked memory address, Pathfinder records the stack trace of the writer. When another thread reads from the same address and so an ITCE is detected the reader thread's stack trace is recorded. Secondly, in addition to the stack trace, Pathfinder will record the *memory identifier* to link the memory address back to a specific field or array declaration in code:

- If the *memory identifier* is a static or global field, we record the code location of the field. That is *package name* of the class, *class name* and *field name*.
- If the *memory identifier* is an array, so the write happens to the allocated space of the array, we report the code location of the last accessed global/static field pointing to that array. If the array is exclusively reachable through a position in another array we report the location as unknown. The array can be used at different places in the application. The location we record is approximativ and can be one out of many.

4.4.3 Pathfinder Reports

The output of the tracking stage is a **Pathfinder Report**: a file that is generated during the tracking stage that contains communication slices for all detected ITCEs. The Pathfinder report serves as input to the investigation stage. In the following we give a brief overview about how instrumentation can be derived from communication slices. We present more on how Pathfinder supports the investigation of Pathfinder Reports in Chapter 5.

4.5 Request-Guided Instrumentation

4.5.1 Deriving Instrumentation from Communication Slices

Communication slices identify where developers should investigate in the code, and where the constituent parts of some higher-level execution pattern may be executing. Often, these places are scattered across several class and method definitions. The slices clearly identify which pieces are related to one-another. During the investigation

stage, developers look at the data of detected slices to infer the execution pattern and derive instrumentation. In the example above, the developer would investigate `compute`, `process`, and `processingLoop`, and finally derive the instrumentation by identifying places for `fork()`, `join()`, `discard()` and where to transfer the context, such as piggybacking the *Work* object here.

Pathfinder is *overcomplete*: many memory addresses might exhibit an ITCE without being relevant to instrumentation. For example, incrementing a concurrent counter will involve reads and writes, though there may be no relationship to the thread that previously updated the counter. Updating metadata in shared data structures, such as filesystem metadata, will have an ITCE with previous threads that wrote the metadata, even if they are not part of the same request. Threads might lazily initialize global system infrastructures including long living objects and data-structures. As part of requests threads access these common data-structures what results in ITCEs between them and the initialization thread. Those ITCEs are not interesting for context propagation. Pathfinder is able to filter out certain irrelevant slices; we discuss this more in §5.4.

There is no strict deterministic pattern for how to identify execution patterns from communication slices (we demonstrate this in our evaluation in Chapter 7). Consequently, Pathfinder leaves this to be a manual step for developers to apply their own intuition. We consider automating instrumentation to be a significant challenge and beyond the scope of our work. In this work we are only interested in capturing and exposing communication slices for *developers* to then reason about; we leave it to future work to explore automatically deriving instrumentation from these inputs.

Consequently, a key goal for the investigation stage is to present developers with the most relevant data to enable rapid progress. When we apply Pathfinder to an entire system, we still encounter a large volume of detected communication slices. Many of these communication slices are redundant – that is, there might be numerous slices that correspond to just one communication channel. This occurs commonly – in the previous example, we detect a communication slice for each field of the Work object, even though we only need to propagate context once. In the next chapter we will describe several techniques Pathfinder provides for filtering data when presenting it to developers during the investigation stage.

4.5.2 Scoping Instrumentation to Requests

Our third intuition thus relates to *which* information we collect and present during each iteration of the tool. Pathfinder could record every communication slice exhaustively throughout the program. This would present an information overload to users; even though instrumentation eventually doesn't touch much of the code base, the developer would still need to sort through many communication slices to identify instrumentation points. In each tracking run of Pathfinder the target system is run. While the system operates many system internal operations and requests execute that all involve ITCEs that cause communication slices that would need to be investigated.

To address this, Pathfinder is based on building up instrumentation successively along code paths of single requests. This follows the established recommendations of existing distributed tracing documentation (cf. §3.1.1). To build up system instrumentation a developer instruments each request type one-at-a-time. The developers only need to consider the request types they are interested in. Pathfinder will only detect communication slices relevant to these request types of interest. The instrumentation process for a single request may involve several iterations. In each iteration, Pathfinder tracks inter-thread communication slices which are then used by a developer to decide where to put instrumentation code. Based on the new instrumentation code the developer inserted, Pathfinder can infer where the request continues and so extends the tracking accordingly. So in each iteration Pathfinder reveals more and more execution boundaries that appear as part of request executions until all are revealed and so all request paths are instrumented.

From a developer's perspective, this approach has several intuitive advantages. It substantially prunes the space of communication slices that developers must consider in each iteration. This is of key importance to make instrumentation a tractable task. However, it does so in a logical way, instead of just selecting an arbitrary subset of the code base. Even though it only tracks a subset of the system's source, Pathfinder will nonetheless track encountered execution patterns coherently – i.e., when one is encountered, all communication slices will be tracked. Furthermore, the detected slices will have some degree of overlap in the writer and reader stacks, differing only in the last few methods and field locations. This further reduces the cognitive overhead for developers.

4.5.3 Tracking Scope API

Pathfinder is an iterative tool for gradually building up instrumentation in a system. Our approach for scoping instrumentation to requests is to *make use of the partial instrumentation* for guiding where the tracking stage should

next apply.

Pathfinder provides a *tracking scope API* for developers to define where a request starts and along which code paths it advances. This API directly corresponds to the general instrumentation interface in §2.4.2. By using the tracking scope API, Pathfinder can restrict itself to just a subset of all trackable memory addresses. This reduces the computational overhead of tracking, but moreover, reduces the amount of output a developer must interpret in each iteration. At the end of the whole Pathfinder process, the instrumentation inserted with the tracking scope API will have fully identified all instrumentation points.

Tracking Scope During the tracking stage, Pathfinder will only track *writes* to memory addresses when they occur within a *tracking scope*. Reads are tracked independent of tracking scopes. Developers demarcate tracking scopes in the system's source code with instrumentation; first during the setup stage, and then during the instrumentation stage for each iteration of the tool.

`TrackingScope.begin(tag)`	Marks the beginning of a tracking scope. Subsequently, any writes to trackable memory addresses performed by the thread that enters the tracking scope will be tracked. `tag` is an identifier that relates the scope to the source-code location of this invocation.
`TrackingScope.end()`	Marks the end of a tracking scope. Writes will no longer be tracked.

During the setup stage, the first task of the developer is to choose an appropriate place to begin and end a tracking scope. The most sensible choice is on the client side where application API calls are initially triggered or at the RPC entry point to the process where an API call is started to be processed. In practice this is easy to identify, and Pathfinder only requires one tracking scope to start with for one request.

During the tracking stage, Pathfinder internally creates some metadata when a tracking scope begins. This metadata includes a unique *scope ID* for the scope, plus the specified *scope tag* to relate the scope to the specific tasks that is executed by the thread. Tracked communication slices can later be grouped based on this tag.

Following Request Paths Pathfinder only detects *writes* within a tracking scope, but will detect all reads. Thus the output of any iteration of Pathfinder will provide one additional level of *possible* request flow to other threads.

During the investigation stage, developers will identify places where instrumentation should go. Pathfinder's tracking API provides methods for developer to extend tracking across a thread boundary. The API corresponds to the general `fork` and `join` instrumentation API described in §2.4.2.

`TrackingScope.getContext()`	If called within a tracking scope, returns a `TrackingScopeContext` containing a copy of the tracking scope's metadata; null otherwise.
`TrackingScope.join(ctx, tag)`	The receiving thread *inherits* the provided `TrackingScopeContext` as follows: • If the provided context is null, the operation has no effect. • If the receiving thread has no tracking scope, then a new tracking scope will begin, but using the provided metadata. • If the receiving thread is already within a tracking scope, then the provided metadata will be merged with the current metadata. If the scope IDs are different, both are retained. The new provided scope tag is ignored, and the present scope tag from the existing metadata is retained.

Using this API, instrumentation can gradually follow a request path, through successive iterations of Pathfinder. Initially, a developer only needs to choose a starting point in the system to begin investigating. In each iteration of the tool, the developer will extend the instrumentation by one "hop", unlocking new tracking points in other threads. Eventually, after several iterations, the instrumentation is complete. We cover this process in §4.6.

4.5.4 Example

We demonstrate the API with two snippets from our running example.

The initial TrackingScope instrumentation is shown in Listing 4.2. During the setup stage, the developer identifies `startRequest` as the place to begin investigation, and creates a Tracking Scope. The developer proceeds to the tracking stage.

```
1  Queue<Work> workQueue = new Queue<Work>();
2
3  class Work {
4    String name;
5  }
6
7
8  // executed by T1
9  startRequest(Request q) {
10   try {
11     TrackingScope.begin("startRequest");
12     compute(q.name);
13   } finally {
14     TrackingScope.end();
15   }
16 }
17
18 compute(String name){
19   doWork();
20
21   Work w = new Work();
22   w.name = name;
23
24
25   workQueue.put(w);
26   doMoreWork();
27   w.await();
28 });
29
30 RunThreadsWith(processingLoop() {
31   // executed by T2 and others
32   while (...) {
33     Work w = workQueue.take();
34
35
36     process(w.name, w); // accessing Work.name
37     w.signalDone();
38
39   }
40 });
```

Listing (4.2) The initial instrumentation is added to the `startRequest` method.

```
1  Queue<Work> workQueue = new Queue<Work>();
2
3  class Work {
4    String name;
5    TrackingScopeContext context;
6  }
7
8  // executed by T1
9  startRequest(Request q) {
10   try {
11     TrackingScope.begin("startRequest");
12     compute(q.name);
13   } finally {
14     TrackingScope.end();
15   }
16 }
17
18 compute(String name){
19   doWork();
20
21   Work w = new Work();
22   w.name = name;
23   w.context = TrackingScope.getContext();
24
25   workQueue.put(w);
26   doMoreWork();
27   w.await();
28 });
29
30 RunThreadsWith(processingLoop() {
31   // executed by T2 and others
32   while (...) {
33     Work w = workQueue.take();
34     TrackingScope.join(w.context);
35
36     process(w.name, w); // accessing Work.name
37     w.signalDone();
38     TrackingScope.discard();
39   }
40 });
```

Listing (4.3) After inspecting communication slices during the tracking stage, new instrumentation is added to `compute` and `processingLoop`.

Figure 4.2: Code listings illustrating a developer's use of Pathfinder's tracking API for two iterations of the Pathfinder tool. The modifications made by the developer in each iteration are highlighted in blue. For each iteration, the developer adds instrumentation. During the next tracking stage, Pathfinder uses the existing instrumentation to identify new communication slices for the next investigation stage.

During the tracking stage, Pathfinder will track all writes within the `compute` method, and all reads performed by any thread to capture the three communication slices listed previously in Table 4.1. Pathfinder will *not* track any writes elsewhere; for example, writes starting in `T2`.

During the investigation stage, the developer identifies that the `Work` object should be instrumented, and extends the TrackingScope instrumentation as shown in Listing 4.3. The developer can proceed to another tracking stage.

During the second tracking stage, Pathfinder will propagate the tracking scope from `T1` to `T2`. When `processingLoop` executes, it will use the propagated tracking scope and Pathfinder will track possible communication slices that appear between `T2` and `T1` or any other thread that communicates with `T2` while processing the work item.

We provide in Appendix B a more extensive example, demonstrating the TrackingScope API usage.

4.5.5 Process Boundaries

Request tracking is primarily used in distributed systems. To enable tracking across process boundaries, we provide a serialization API so that tracking scopes can continue on the receiver side of RPCs.

`TrackingScopeContext.serialize()`	A member method of `TrackingScopeContext` that serializes the context and returns a byte array.
`TrackingScope.deserialize(byte[])`	Deserializes a byte array and returns a `TrackingScopeContext`.

Pathfinder does not intentionally help identify the process boundaries in code where these API calls should be inserted. Pathfinder focuses on inter-thread communication and does not detect when threads access sockets or files. Pathfinder relies on the developer to identify these places manually. Typically, inter-process communication is more well defined and easier to identify than intra-thread communication, occurring through well defined, narrow interfaces.

4.5.6 Incorporating Existing Instrumentation

Pathfinder is able to take existing instrumentation into account. In order to incorporate existing instrumentation, existing calls to a request context propagation API must be aligned with calls to our tracking scope API presented in §4.5.3. TrackingScopeContext must be propagated along with existing request context flow. Since the tracking scope API directly corresponds to common context propagation tools, this just requires to combine or replace existing calls of `create`, `fork`, `join`, `discard` with `TrackingScope.begin`, `TrackingScope.getContext`, `TrackingScope.join` and `TrackingScope.end`.

4.6 Summary

In §4.2 we outlined the main stages involved with using Pathfinder. We conclude this chapter with a summary of the key design details of each stage.

To reach a system instrumentation a developer selects a set of requests for which end-to-end context propagation should be implemented. Each of these requests is then instrumented using Pathfinder.

To prepare the application of Pathfinder, inter-process tracking scope propagation should be implemented by the developer using the API presented in §4.5.5. Pathfinder focuses on inter-process context propagation and so is not able to detect process boundaries where tracking scopes must be serialized and deserialized. With this instrumentation in place tracking is automatically continued on other processes when a request crosses a process boundary.
After this preparation the developer proceeds as follows for each request.

During the one-off **Setup Stage**, the developer chooses a starting point for Pathfinder to begin, and creates a TrackingScope. The standard starting point would be a request start point at the client side or an API call entry point.

In the **Tracking Stage**, the developer runs a workload with Pathfinder attached to the system. Pathfinder will statically analyze the code to identify fields and arrays that should be tracked; then at runtime, it will track reads and writes to those addresses. Only writes occurring within a TrackingScope will be tracked. For all detected ITCEs, Pathfinder will record a communication slice comprising a reader stack trace, writer stack trace, thread IDs, the memory identifier, the tracking scope ID, and the tracking scope tag. The output of the tracking stage is a **Pathfinder Report** that records all detected communication slices.

The **Investigation Stage** comes next; the developer uses a tool we have developed called **Pathfinder Explorer** to investigate communication slices and decide where to insert new instrumentation. In the next chapter we describe Pathfinder's exploration features.

Next, during the **Instrumentation Stage**, the developer adds new TrackingScope calls to extend the reach of Pathfinder to other threads. During the subsequent tracking stage, Pathfinder will propagate its scope metadata to these new threads, creating new TrackingScopes, and resulting in new communication slices in the next tracking run.

If the investigation stage did not point to any new instrumentation points, the developer drops out of this loop, because all instrumentation points were discovered for this request type. The developer may then choose to instrument a different request type, or end here.

When the system is instrumented, the tracking scope API calls that were added to the system can be replaced with actual calls to a request context propagation API provided by the desired request flow tracking tool. Since Pathfinder's tracking scope API directly corresponds to common context propagation APIs, this task is straight forward.

Chapter 5

Pathfinder Explorer

In this chapter, we describe Pathfinder Explorer, an additional tool we have developed specifically for analyzing Pathfinder Reports. The tool is used by developers during the **investigation stage** to group, sort, filter, and visualize communication slices.

The focus of this chapter is how Pathfinder Explorer presents communication slices to developers. Pathfinder does not simply display the raw report to developers, since Pathfinder reports can be quite large: many communication slices represent the same locations in code; and there might be a large amount of redundant information. Thus, we have developed several techniques to filter, prune, and ultimately visualize Pathfinder's output.

5.1 Goals

A developer investigates Pathfinder Reports to answer the following questions:

- Does a collection of detected communication slices correspond to an execution pattern where instrumentation might be needed?
- How should the instrumentation of this execution pattern be correctly implemented?

To answer the first question developers need to understand what the communication is about, why it happens and under which task it happens. To answer the second questions it is helpful to understand which resources (e.g. work item) are transferred between the threads to understand along which resource flow the request context might be propagated. An example was shown in. In an example we showed (Listing 4.1) the context is forked on creation of a work item, attached to the work item, retrieved from the work item.

5.2 Pathfinder Report

Typically, the detected communication slices are somewhat localized in the code. This occurs because of Pathfinder's tracking scope, which restricts the attention to only a subset of the code base. Figure 5.1 illustrates a hypothetical example where a request has communication between four different threads; it shows which communication slices are detected in each iteration. In this example Thread `T1` started the execution of the request. In the first iteration `T1` communicates with `T2` and `T3`. Corresponding ITCEs were detected, and the communication slices were captured. All of the captured communication slices carry the same scope tag `TAG1`, because the tracking scope originated from the same line of source code.

A developer examines the output and identifies the communication between `T1` to `T2` as relevant so adds new TrackingScope instrumentation. The communication between `T1` to `T3` is identified as irrelevant, so no instrumentation is necessary there. In the subsequent iteration, scope metadata is propagated from `T1` to `T2`. New ITCEs are now detected: `T2` communicates with `T4`. In this second iteration of the tool, the developer's attention shifts to corresponding captured slices with scope tag `TAG2`.

5.3 Pathfinder Explorer

We present how communication slices can be investigated to answer the question mentioned under goals.

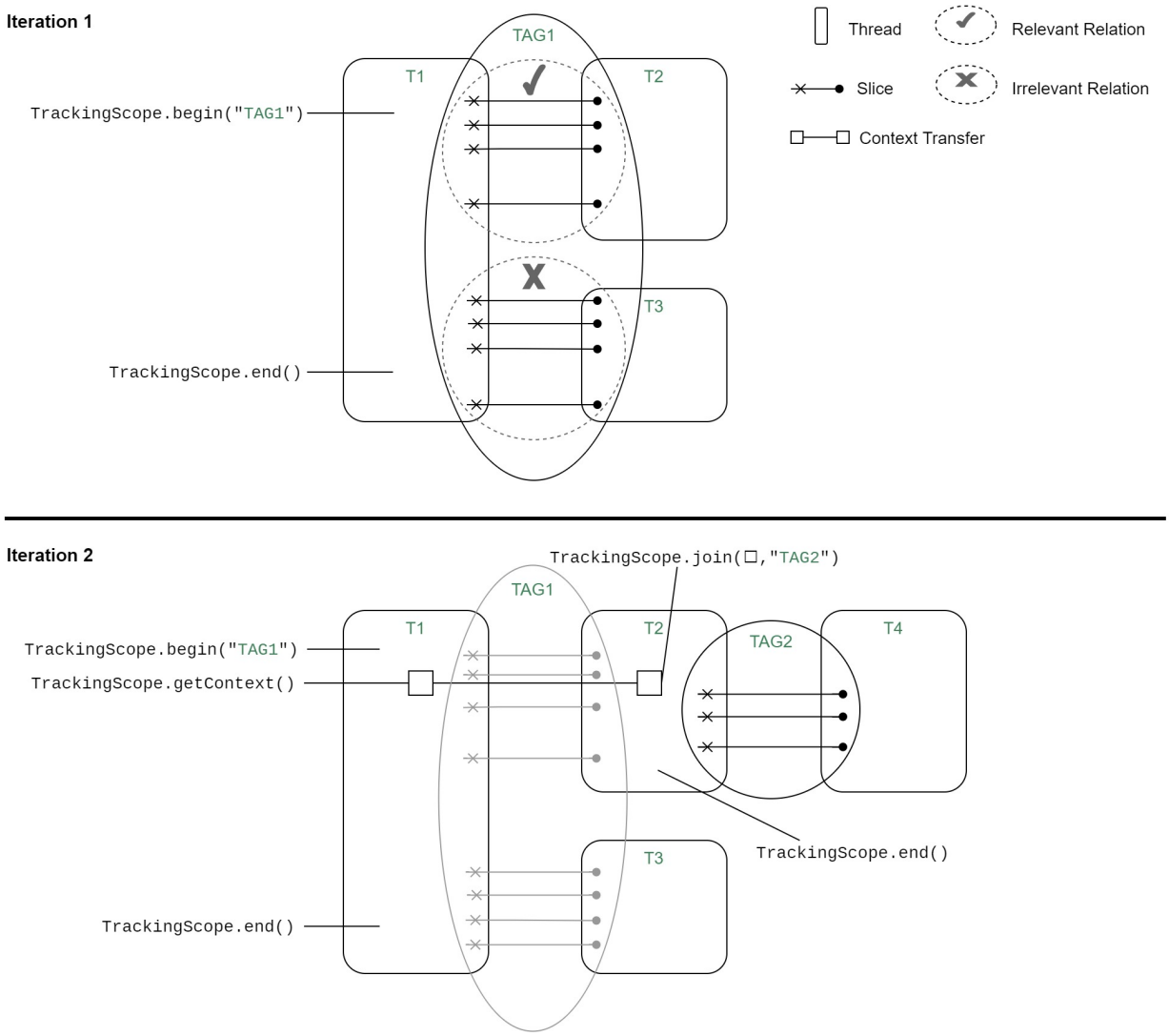

Figure 5.1: Communication slices detected during two iterations of Pathfinder.

Visualization Pathfinder Explorer provides several visualizations for investigating individual communication slices, as well as communication slices in aggregate. Developers can select a specific subset of slices to investigate according to their own filtering criteria, before then visualizing the slices. We provide examples momentarily.

Filtering Pathfinder Explorer includes several filtering techniques for reducing the number of communication slices. We can heuristically identify that many slices are not relevant to investigation through post-processing, and filter these out before displaying information to a developer. We describe these techniques in §5.4.

Thread Pairs The main dimension along which we partition a Pathfinder Report is a *thread pair*. One thread pair consists of a writer and a reader thread, identified by the thread ID included with each communication slice. In one iteration we investigate all communication slices that belong to one thread pair at a time. Referring again to the example in Figure 5.1, in the first investigation stage, the developer would separately investigate communication slices between `T1` and `T2`, followed by communication slices between `T1` and `T3`. Similarly in the second investigation stage, the investigation would proceed with `T2`, `T4`.

Scope Tags In addition to the thread IDs, communication slices can be partitioned by scope tag, which identifies the line of code that created the tracking scope.

5.3.1 Investigating Single Slices

The developer iterates through the communication slices until she can deduce what the communication is about and if any context propagation is necessary. Anecdotally, we have found that only a small number (e.g. 5) of communication slices are needed before an execution pattern and corresponding instrumentation can be deduced.

To investigate a communication slice, the most important information to consider is the relationship between the writer and reader, and through which memory identifier. By inspecting a slice, the developer can often identify subsequent methods of interest to investigate.

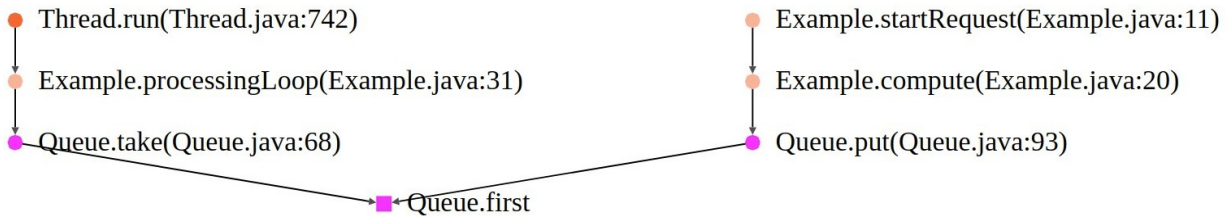

Figure 5.2: Slice visualization. Reader stack on the left side. Writer stack on the right.

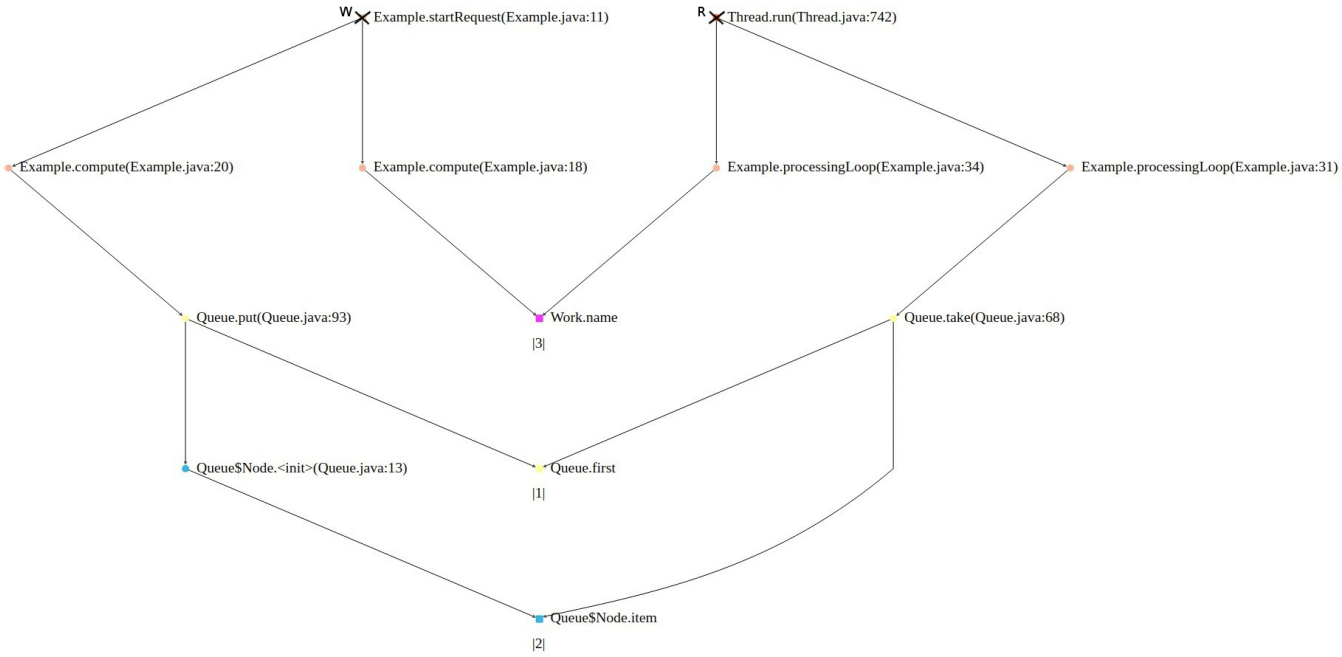

Figure 5.3: Visualization of multiple slices. Reader and writer stack traces are merged. Stack trace start points for reader and writer are marked with a cross. Numbers underneath presented fields indicate the reader access order.

For example, consider the slice depicted in Figure 5.2, corresponding to one detection from our running example (cf. Figure 4.2).

5.3.2 Investigation Order

The default approach we take to investigation is to inspect individual communication slices in order. Pathfinder presents slices in reader access order. The communication slice that is first accessed by the reader is presented first. This often points to the root point of the communication. For example, when a work item is passed to another thread, this thread first accesses the data structure over which the work item is passed, or the meta data that is stored in the work item. These accesses happen close to the initial communication point. The work item's payload, which fields are often written first, are usually read later when they are used as part of actual processing logic. In this case the stack traces do not contain the methods that point to the communication point what makes it harder to understand how the message actually reached the reader thread. But this is what we are most interested in for the instrumentation.

5.3.3 Visualization

Pathfinder Explorer provides several visualization features for investigating single communication slices or multiple communication slices at the same time. Figure 5.2 depicts a visualization for a single communication slice. Figure 5.3 depicts a visualization for all three communication slices from our example. By depicting multiple slices simultaneously, developers can further deduce the relationships between the multiple memory locations. In the example we can see that a common prefix of all slices is the `processingLoop` and `compute` methods, suggesting this is a good place to investigate.

We provide in Appendix A several more examples of real-system slices detected during our evaluation.

5.4 Filtering Communication Slices

We now describe several filtering techniques used by Pathfinder Explorer to reduce the number of communication slices for investigation. Some filtering is entirely local and is performed during the tracking stage; the rest occurs as a post-processing step in Pathfinder Explorer.

5.4.1 Irrelevant Slices

For various reasons, Pathfinder might record a communication slice that is not relevant. This means they are not helpful for a developer during the investigation stage. Here we describe several categories of slices that are irrelevant, that Pathfinder filters out.

Null Values

Description: ITCEs are irrelevant when writing and reading null values. For example when the last entry of a linked list is removed, the field pointing to the last entry is set to null. Under the task of iterating through a list, reading this null value results in an ITCE if the thread that removed a last entry is a different one than the reader. A similar case appears when objects are reused and cleared before returning to a pool.
Filter: To eliminate such ITCEs we do not track the writes of null values. This occurs during the tracking stage.

Runtime Internals

Description: There is implicit communication between runtime internal threads and application threads, e.g. when a thread of the runtime garbage collection finalizes objects and thereby reads fields of that object.
Filter: To eliminate corresponding communication slices we filter based on the stack traces. The garbage collector executes runtime specific code. Pathfinder removes all tracked slices where the reader stack trace contains any method of runtime internal code packages such as the garbage collection package. This occurs as a post-processing step.

Thread Synchronization

Description: The usage of synchronization mechanisms like locking can lead to ITCEs. When a lock is taken or released writes and reads by different threads are involved. Intuitively, synchronization operations should arise when information exchange takes place, because they fundamentally protect shared data; prior works have noted and exploited this [23]. However, the synchronization mechanisms themselves do *not* help indicate *which* information is exchanged. Moreover, Pathfinder will detect the actual information exchange that occurs within the critical section. Thus, tracking the synchronization mechanisms is superfluous and is sometimes a red herring. For example, entering a lock when a thread polls a work item from a queue does not imply that the thread that entered the lock before submitted this work item. A similar argument holds for signal/wait relations. Signaled threads might poll work items that arrived after receiving the signal. Unless one is not particularly interested in lock/release relations and synchronization events, context flow along these events is unnecessary.
Filter: The usage of locks, conditionals and signal/wait involves specific classes and internals. During the tracking stage, Pathfinder provides the option to disable the observation of writes and reads for most of these components. Pathfinder also provides the option of filtering based on stack traces as a post-processing step.

Class Loading

Description: Many runtimes load classes lazily when they are first required. The first thread that needs to use a class will perform the class loading; afterwards other threads will use the loaded instance. This leads to read-after-write

events between those threads through class loading infrastructures. These ITCEs do not correspond to any application logic and so are irrelevant.
Filter: Pathfinder pauses tracking of ITCEs while class loading during the tracking stage.

Duplicates

Description: Multiple ITCEs can end up producing equal communication slices, if they have the same stack traces and access the same memory modifier. This is possible if several message objects are exchanged between threads in a loop or if multiple invocations of the same code executes in parallel threads.
Filter: Pathfinder eliminates redundant communication slices as a post-processing step. First Pathfinder eliminates all duplicate communication slices with identical thread IDs, memory identifiers and stack traces.
Second Pathfinder considers duplicate communication slices that only differ in their thread IDs. Removing these duplicates is useful for developers, but we must be careful to do so consistently. Then we consider all thread pairs of one writer thread (thread appears as writer). We remove all slices corresponding to one thread pair if there exists another pair having an equal set of slices. Slice sets A and B are considered equal if for each slice in set A there exist an slice in set B with the identical memory identifier and stack traces and for each slice in set B there exist an slice in set A with the identical memory identifier and stack traces. This operation is not equivalent to just iterating through the full slice list and removing equivalent ones. Such an operation can cause inconsistencies when removing not one thread pair as a whole but slices across thread pairs.

Cross Request

The parallel execution of requests or system internal operations can cause ITCEs between threads that belong to different requests/system operations. For example through global caches or state. The relevance of corresponding communication slices is use case dependent.
Filter: To minimize such ITCEs only one request should be executed on the system at a time. Accordingly, requests should be executed in parallel or subsequently if such cross request relations are relevant.

The execution of system internal operations which are not user requests is usually hard to control. Pathfinder does not provide additional mechanisms to further control which ITCEs are tracked that appear between tracked requests and system internal operations. We rely on the developer to filter out remaining slices that are irrelevant.

Initialization

When a request is first executed it might initialize required infrastructures what can lead to ITCEs between the initialization thread and other threads that use the infrastructures. These ITCEs do not correspond to data in transit and are usually irrelevant.
Filter: To prevent Pathfinder from tracking those ITCEs, a workload should be executed on the system before tracking is started. This workload will trigger the initialization independent of any tracking run.

5.4.2 Increasing Variance of Communication Slice Sets

To further speed up the investigation process, Pathfinder Explorer removes communication slices that are highly similar to others, reducing the amount of redundant information a developer has to inspect.

Concretely, Pathfinder Explorer filters redundant communication slices with memory identifiers that are members of the same parent object. Pathfinder does the following: To begin, the list of slices is sorted accordingly to the reader access order starting with the earliest access. We then iterate in reverse order over this list and eliminate a slice A if there exists another slice B in iteration direction with A.parent = B.parent and the stack trace of the reader respectively writer of A equals the stack trace of the reader respectively writer of B *excluding* the line number of the most current stack trace entry.

slice.parent is defined as follows. If the slice corresponds to an

- array index access – slice.parent is the reference to the array
- static field – slice.parent is packagename + class name of the memory identifier
- global field – slice.parent is a reference to the parent object the field is a member of

Since the stack traces are almost identical, we loose no crucial information here. We remove slices from the end to the start and so keep early slices which might correspond to the initial communication point. This technique also eliminates identical slices that might be caused by a loop execution.

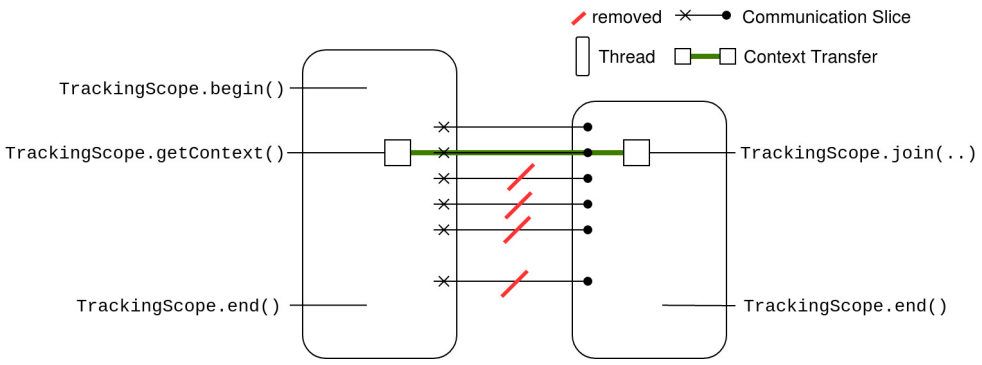

Figure 5.4: Remove Covered Slices

5.4.3 Covered Inter-Thread Communication Events

After we instrumented an inter-thread communication channel with Pathfinder we do not want to investigate communication slices that correspond to the same inter-thread communication channel again. But this is what will happen by default. Different requests share the same components, so their executions will traverse the same inter-thread communication channels several times.When we start a tracking scope for a request, Pathfinder tracks all ITCEs that follow, even if they correspond to covered instrumentation points. Communication slices that belong to the same boundary are tracked again and the developer has to identify them as redundant and skip them. Doing this over and over for different requests is time consuming.

To address this we provide a mechanism to automatically identify and remove communication slices that belong to inter-thread communication channels that were already covered.
The usage of Pathfinder involves propagating TrackingScopeContexts along the same paths the request context would take. We exploit this fact. We automatically infer the TrackingScopeContext flow and so identify if a slice occurs between threads that already exchanged the TrackingScopeContext. If this is the case we safely remove corresponding slices.

Eliminating Covered Communication Slices

We realize this automatic elimination of slices as follows. We associate the tracking scope ID, each tracking scope carries, with each memory address that is written by a thread under a tracking scope. When a tracking scope is merged into another using TrackingScope.join(), the tracking scope ID is added as a parent tracking scope reference. When a communication slice is captured, Pathfinder captures the writer tracking scope ID and the list of parent scope IDs of the reader's tracking scope, if there is any. Before slices are investigated by the developer, a slice is removed if the writer's tracking scope ID is contained in the list of parent tracking scope IDs of the reader. We illustrate which slice can be removed this way in Figure 5.4.

Slices that are captured before TrackingScope.join() or after TrackingScope.end() usually appear as part of using a data structure to exchange messages. For example objects that wrap list entries are written and read before the actual message and context. To eliminate them we provide an additional option to remove all slices that appear between threads that triggered at least one ITCE which slice could be removed with the presented technique.

Verify Instrumentation

This feature can also be used to verify the effects of context propagation. We can identify all ITCEs that appear after the TrackingScopeContext was joined and before it is ended and also which ITCEs appear before TrackingScope.join() and after TrackingScope.end(). With this information a developer can verify if the context flow covers all code areas it should.

Chapter 6

Implementation

We have implemented Pathfinder for tracking ITCEs in Java programs. We have also implemented Pathfinder Explorer for presenting Pathfinder's output. We choose Java due to its widespread use in large-scale open-source distributed systems.

6.1 Overview

Pathfinder uses several techniques to implement tracking for Java programs. Pathfinder's Rewriter is a Java agent that intercepts all class loading and rewrites classes to add Pathfinder's tracking code. The Rewriter analyzes the class and inserts tracking code for fields and arrays. The inserted tracking code comprises invocations of Pathfinder's AccessTracker. The AccessTracker is an application-level component that maps memory addresses to partial communication slices. Lastly, a background Logger component outputs the captured communication slices. We describe each component in more detail.

6.2 AccessTracker

6.2.1 TrackingScope

We implement Pathfinder's TrackingScope with the API described in §4.5.3. When `TrackingScope.begin` is invoked, a scope object is created with a unique ID and the given tag. The object is stored in a thread-local variable.

6.2.2 API

Pathfinder's AccessTracker component is invoked whenever tracked memory addresses are read or written. Its main responsibility is to store partial communication slices when memory addresses are written. It has the following API:

`AccessTracker.write(location, value)`	Invoked after the current thread has performed a write to the specified memory `location`
`AccessTracker.read(location)`	Invoked after the current thread has performed a read from the specified memory `location`

Internally, Pathfinder's AccessTracker maintains a map from `location` to partial communication slice information. An invocation of `write` will capture the writer-side portion of the slice (thread ID and stack trace) then store this information in the map, using `location` as the key, overwriting any previous mapping to `location`. This only occurs if `value` is not `null`, and if there is an active tracking scope. If `value` is null then any previous mapping to `location` is simply removed.

An invocation of `read` will use the map to look up if there was a previous write to the same `location`. If not, then nothing further occurs. If there was a previous read, then the AccessTracker compares thread IDs to ensure this was an ITCE. If so, Pathfinder will capture the reader-side portion of the slice (thread ID and stack trace) then enqueue the completed slice to be output by an asynchronous logging thread.

6.2.3 Memory Locations

Pathfinder uses `java.lang.System.identityHashCode()` from Java's runtime in lieu of concrete memory addresses. `identityHashCode` is designed for purposes such as ours – it provides an ID that is likely to be unique for instances of Java objects. However, it alone is not sufficient for our purposes. We explain by example. Consider the following snippet of code:

```
1  class Work {
2    String name;
3  }
4
5  Work w = new Work();
6  w.name = "MPI";
```

In the above, suppose `name` is the field through which an ITCE is detected. To get the address of `name`, it is incorrect to use `System.identityHashCode(name)`, as this returns the address of the *value* of `name`, because objects in Java are pass-by-reference. Instead, the address of `name` is in fact some offset of the address of the Work object `w`.

With this intuition, Pathfinder identifies memory locations using the `identityHashCode` of the *parent* object, plus additional information to identify the offset within the parent object.
For static fields, the parent object is null. We just use the fully qualified name of static fields for identification. For array accesses we use the `identityHashCode` of the array object, plus the array index that is accessed. To identify global fields within an object, we use the `identityHashCode` of the object, plus the fully qualified field name.

Because memory locations are identified with field names these can directly be used as the memory identifiers that Pathfinder reports for communication slices. To derive the memory identifier for arrays Pathfinder also has some special-case handling for array aliasing.

6.2.4 Limiting Communication Slice Tracking

To reduce overhead Pathfinder provides an option to limit the tracking of communication slices. The amount of slices that are captured for a thread pair can be limited to a fixed number (e.g. 50). For further ITCEs no slice is captured what safes the time needed for capturing the reader's stack trace. As we will present in the evaluation, it is usually enough to investigate the first few tracked communication slices, so it is not necessary to capture a communication slice for all occurring ITCEs. We implement this feature by maintaining a map that maps thread pairs (i.e. thread ID of writer and reader) to the number of tracked slices. If the specified limit is reached, no further slices are tracked.

6.3 Rewriter

Pathfinder provides a Rewriter to rewrite Java classes. Pathfinder's Rewriter scans Java classes and identifies all lines of code that read or write static/object fields and arrays. The Rewriter inserts invocations to `AccessTracker.write` and `AccessTracker.read`. During this step, the Rewriter essentially has access to the program source, so can construct memory location identifiers using field names in a straightforward way.

Pathfinder's Rewriter is deployed as a Java agent using `java.lang.instrument` [63] to rewrite classes at runtime when they are loaded and as static rewriting tool to rewrite classes ahead of time.

ITCEs can occur while the execution of application code, Java standard or other third-party library code. Pathfinder rewrites all code sources. The Java standard library contained in Java's "rt.jar" is rewritten statically, ahead of time. The instrumented rt.jar must be given as a boot classpath option. Application code and other third-party libraries are rewritten at runtime. The Rewriter uses the bytecode engineering toolkit Javassist [64] to rewrite classes.

In the following we present some special cases and refinements for the default rewriting mechanism.

6.3.1 Immutable Objects

Several Java library classes are immutable and do not need to be rewritten, for example `java.lang.Integer` and `java.lang.String`. We manually identify these classes and exclude them from rewriting.

6.3.2 Lambda Expressions

ITCEs can occur through the *captured variables* of a lambda expression. Lambda expressions in Java are not compiled to bytecode ahead of time such as other code but at runtime when those are first accessed. The bytecode that is

generated does not pass Java's instrumentation framework (i.e. `java.lang.instrument`) Pathfinder relies on. So lambda expressions would not be matched by Pathfinder's main rewriting approach. We address this in the following way: When byte code is generated for lambda expressions at runtime we redirect generated code to our dynamic Rewriter. We implement this by statically instrumenting the lambda factory code contained in the rt.jar.

6.3.3 Native Memory Accesses

Besides standard field and array accesses, Java provides utility classes to do common operations faster via native code or to bypass Java's memory management. Such functionality is provided by the classes `sun.misc.Unsafe`, `java.lang.System` and over Java's native interface (JNI) [65]. Because such operations call native code that performs the write or read, our standard Java byte-code rewriting misses these places. We provide special refinements to capture memory accesses via `Unsafe` and `System`. Memory accesses that happen through the JNI are not observed by Pathfinder.

Unsafe

Pathfinder redirects calls to `Unsafe` to our tracking component where we track the access and perform the actual call to `Unsafe`. We do not redirect all `Unsafe` calls. We only redirect the ones which are frequently used in common data structures under `java.util`. We consider the use of `Unsafe` outside of java internals as rare. We handle the following methods:

- `Unsafe.compareAndSwapObject(Object o, long l, Object a, Object b)`
- `Unsafe.putOrderedObject(Object o, long l, Object val)`
- `Unsafe.putObject(Object o, long l, Object val)`

Correctly tracking and updating memory locations for `Unsafe` code requires some additional consideration. Some `Unsafe` calls work on memory offsets (e.g. `Unsafe.putOrderedObject`). These methods require to provide the parent object of the field, the offset of the target field in that object and the value that should be set. To get the memory offset, `Unsafe.objectFieldOffset` is used. The offset is defined per class, so is instance independent. To refer memory offsets back to actual field code locations we keep track of fields and their offsets that are retrieved via `Unsafe.objectFieldOffset`. On the use of another `Unsafe` method we lookup which field actually corresponds to the memory address for the given object class.

System.arraycopy

Java's `System.arraycopy` is an efficient call to native code for performing bulk array copies. Pathfinder does not observe *read* accesses as part of `System.arraycopy()` that are performed by threads that do not act under an active tracking scope. We do this to avoid additional overhead. Tracking those accesses would introduce significant overhead for *all* array copy operations in the system. We consider missing an instrumentation point because of not tracking those memory accesses as unlikely because of the following reasons:

- We investigated the occurrences of this instruction in Cassandra and HBase. The instruction is mainly used by array based data structures such as array lists when the capacity is extended or when byte arrays are written to buffers but barely in the context of inter-thread communication. In the case of an array list the method is only called when the data structure is extended what happens non-deterministically when the capacity limit is reached. The appearance of such a call while a request execution is rather rare and as we will show in §7.1.3 the same infrastructures and so the same code is used at several places within and across requests. The likelihood that the array is extended every-time it is accessed across several request executions is rather small.
- Experiments showed that threads usually exchange several data pieces within one communication activity. Tracking further slices corresponding to close by memory accesses can compensate a missing slice.

6.3.4 Static Initialization

Pathfinder does not instrument static initializers to prevent class loader deadlocks. A classloader deadlock appears if class A has a static dependency on class B that has a dependency on class A. For example: The static initializer of A uses B that again uses A but the call to B cannot complete because A is not initialized, yet. Because Pathfinder also instruments common data structures that itself uses, such a scenario can be problematic. Static initializers are called once for a class type. This cannot cause writes that belong to dynamically created objects in transit. So we consider scenarios where not tracking memory accesses in static initializers hides relevant inter thread communication as unlikely.

Chapter 7

Evaluation

In this chapter we present an evaluation of Pathfinder on two widely used distributed systems: Cassandra [26] and HBase [25]. We evaluate the completeness of Pathfinder's instrumentation, we approximate the amount of effort required to perform the instrumentation, and we compare Pathfinder's instrumentation to pre-existing reference instrumentation within each of the systems. Further, we compare Pathfinder with an existing automatic instrumentation approach. We also measure the overheads Pathfinder imposes during the tracking stage. Overall, we show that:

- Pathfinder is able to identify *all* instrumentation points of a reference instrumentation in both Cassandra and HBase.
- Using Pathfinder we identified multiple instrumentation points in HBase that are missing from the reference instrumentation.
- Pathfinder required investigating less than 1% of the code base of either system.
- Pathfinder does not suffer the same pitfalls as the automatic instrumentation approach of OpenTracing.
- Pathfinder's runtime overhead is up to 150× the latency of regular requests, which is within acceptable bounds for runtime tracking to be feasible.

7.1 Experimental Setup

7.1.1 Systems

We apply Pathfinder on Apache HBase [25] and Apache Cassandra [26]. Both systems are large scale distributed storage systems. We picked them because of the following properties:

- active and relevant – They are both actively developed and used by many companies. (e.g. Facebook, Uber, Pinterest)
- open source – They are free for academic purposes and offer white box access what is a requirement of Pathfinder.
- complex – A request execution traverses various execution patterns Pathfinder has to deal with.
- existing instrumentation – Both systems provide an existing instrumentation we can use as a reference.

7.1.2 Deployment Setup

We deploy all systems and run all the experiments in a VirtualBox VM that runs Ubuntu 16.04. The resources allocated for the VM are 16GB RAM and 3 CPU Cores. The VM runs on a machine with 32GB RAM and Intel Core i7-6700K (4x4.3GHz) as CPU.

We deploy the different nodes that are part of the distributed systems as separate processes in the same VM. It is required to ensure that request executions traverse all code paths that they would in a real deployment. However, a full production deployment is not required – only a minimal deployment to ensure code paths are followed. Typically it is enough to deploy a system with only one or two nodes. We describe the exact deployment scenarios in line for each experiment.

Request Type	Iterations	Distinct Slices	Investigated Slices		Covered Classes		Covered Methods		Investigated Methods		Instr. Points Found
			Rel.	Total	Rel.	Total	Rel.	Total	Rel.	Total	
create keyspace	2	4.5k	11	96	42	293	66	559	8	27	1
scan	2	0.5k	12	24	30	96	58	170	6	16	1
insert	1	0.5k	0	4	0	21	0	32	0	0	0
update	1	0.5k	0	3	0	19	0	29	0	0	0
delete	1	0.5k	0	3	0	19	0	29	0	0	0
create table	1	5.5k	0	19	0	104	0	191	0	2	0
drop table	1	5.0k	0	29	0	150	0	260	0	2	0
System	9	17k	23	171	61	386	108	801	14	47	2

Table 7.1: Summary statistics for instrumenting Cassandra with Pathfinder. Request types were instrumented in the order presented. Slices, classes, and methods are *relevant* if they correspond to some instrumentation in the reference instrumentation. See §7.2.2 for a detailed description.

7.1.3 Instrumentation Procedure

For each system (Cassandra and HBase), we used Pathfinder to derive instrumentation. We follow the instrumentation workflow described in this thesis, where we selected several request types to instrument one-by-one. We ran a small workload before tracking a request, to reduce the chance of lazy object initialization triggering spurious ITCEs (we covered this topic in §5.4). We then tracked exactly one request, to minimize the chance of cross-request slices.

7.2 Instrumenting Cassandra

In our first experiment, we use Pathfinder to derive instrumentation for Cassandra. The focus of this experiment is firstly instrumentation completeness: does Pathfinder successfully identify all instrumentation points? To answer this question we compare to pre-existing instrumentation that already exists within Cassandra. That prior instrumentation was added by the system's developers manually, to provide distributed tracing, at great pains (as outlined in Chapter 3). The existing instrumentation serves as a reference for assessing the relevance of Pathfinder's communication slices, and enables us to report on completeness and correctness of Pathfinder's derived instrumentation.

The second goal of this experiment is instrumentation effort. To assess this, we measure several metrics relating to how much of the code base contained ITCEs, was presented by Pathfinder Explorer, and was investigated to implement context propagation. While there is no accurate baseline to estimate the amount of effort for a manual instrumentation, we compare our measured metrics to the overall code base, and show that Pathfinder only requires investigating less than 1% of the overall code base in detail and less than %3 as part of the high-level investigation of stack traces. To provide qualitative context for understanding the results we also report on the wall-clock time required to perform this instrumentation.

7.2.1 Setup

We used Cassandra Version 3.11.6. We ran a two node Cassandra cluster (two processes on the same host), sufficient to exercise Cassandra's primary and secondary node code paths. Having more nodes provides no additional benefits.

We modified the source code of Cassandra to prepare the application of Pathfinder. We implemented inter-process context propagation (the `serialize` and `deserialize` methods) ahead of time in Cassandra's networking layer to allow Pathfinder to track slices across processes. Identifying such code locations is trivial and required code modifications in 4 places. We implemented a small Cassandra client to set the start point of requests and initiate tracking.

We increased the request timeouts on the client side to tolerate the overhead Pathfinder introduces to 20 seconds. Request executions would time out and be aborted under Pathfinder without this extension. The default is 2 seconds.

7.2.2 Results

We instrumented 7 request types, shown in Table 7.1, in the given order. The order is arbitrary, but relevant because instrumentation found for one request type is then already covered for later request types.

Table 7.1 shows how many *relevant* inter-thread slices and *irrelevant* inter-thread slices we investigated across the iterations. A slice is *relevant* if the reference instrumentation contains context propagation along the communication channel the slice points to. Accordingly, a slice is *irrelevant* if there is *no* corresponding context propagation in the reference instrumentation.

We provide a brief description of the overall investigation process. We began with the "create keyspace" request type. A request begins with the client program performing an RPC to one of the Cassandra nodes. The request is received and a thread starts processing: it writes to a commit log, changes in-memory state, and notifies other nodes to replicate the data. The other nodes perform essentially the same operations. When all nodes complete and return an acknowledgment, the operation returns to the client.

Using Pathfinder required 2 iterations to complete the instrumentation for this request. Overall, Pathfinder tracked approximately 4500 distinct (i.e. unique stack trace and memory identifier) slices for this request. Of those only 96 slices required investigating – others were either filtered out (cf. §5.4), or were skipped while the investigation because the necessary information could already be derived from previous slices. 11 slices we investigated were relevant and 85 were irrelevant.

Consequently, we added instrumentation for only one execution pattern: the base class of an execution service that is used in multiple places in the code.

Overall, the communication slices we considered during the investigation named 293 classes and 559 methods – that is, these classes and methods appeared on either the reader or writer stack traces. Often this is enough for us to infer patterns in the code.

Sometimes, further investigation of the actual code (i.e. method bodies) is required. This is necessary to decide the final instrumentation points, or if relationships between threads cannot be fully inferred by the stack traces. This was necessary for 27 of the 559 methods.

Table 7.1 further shows the results for other request types. The entry **System** summarizes the results across all request types.

7.2.3 Derived Instrumentation

Pathfinder was able to identify all instrumentation points that are covered in the reference instrumentation. We verified this based on the reference instrumentation code and produced traces. There are only two instrumentation places because all requests use a shared set of execution services. Instrumenting their base classes was sufficient to cover all requests.

In addition to the reference instrumentation, Pathfinder identified several additional places where instrumentation might be desired. We are unable to assess whether this instrumentation is omitted in Cassandra by mistake, or if it was a qualitative developer choice because distributed tracing is undesirable in these locations. We categorized corresponding slices as irrelevant in Table 7.1. We list such detected places here:

- The reference instrumentation does not associate RPC sending and receiving tasks with requests. The reference instrumentation embeds request context into inter-process messages before passing them to separate threads that perform the actual RPC. So context is not propagated to those separated threads and so the actual sending task that is performed by those separate threads is not captured for a request.
  A more fine-grained instrumentation at this point may have been undesired for distributed tracing.
- The instrumentation does not capture some asynchronous tasks that are triggered by the request, such as a cleanup task. These are unlikely to be desired for distributed tracing, and likely were omitted for this reason. However, it might be reasonable to instrument them as a separate background task, in which case Pathfinder is useful for identifying these starting points in code.
- The reference instrumentation does not propagate context back to callers at places where results of concurrent executions are retrieved like in the example shown in Figure 2.1, introduced in §2.4.2. This is not required for the distributed tracing tool used in Cassandra.
- Cassandra uses a write-ahead log on the critical path of some requests. This code path is not instrumented, but could be included as part of a distributed trace. The instrumentation for this execution pattern is not straightforward because several log operations across several requests are summarized to one write out operation. It is not obvious how to associate the one write out operation with all the requests with the tracing tools Cassandra's instrumentation supports. We believe this is the reason why this place is omitted.

7.2.4 Irrelevant Slices

The results show that most of the slices we investigated were irrelevant. Out of 171 slices only 23 corresponded to eventually relevant instrumentation places. In addition to omitted instrumentation listed above, the remaining slices were caused by system operations that read data the tracked requests had written during the execution including cached data, metric information, global application state (e.g., endpoint information, in memory storage tables) and custom synchronization structures. Pathfinder is not able to filter such slices out. This is one of the main arguments why Pathfinder relies on a developer to do this. For a human it is usually easy to identify a slice as relevant or irrelevant.

7.2.5 Diminishing Returns

As the instrumentation progresses, the number of captured slices reduces in each iteration. This is caused by:

- The elimination of covered slices. After the first two request types, we had identified and instrumented all necessary places in the code base. For subsequent request types, those slices were automatically eliminated with the technique presented in §5.4.3.
- Slices that cover operations that were already fully explored can be filtered out in subsequent iterations. We remove such slices based on method names that belong to those operations and appear on the stack traces. For example, create keyspace and create table both partially use the same code paths.We only remove such "redundant" slices based on operation names if we can be sure that the triggered operation cannot lead to unseen behavior depending on arguments and state.

7.2.6 Tracked vs. Investigated

Table 7.1 shows a significantly higher number of tracked slices than actually investigated. This discrepancy results from the filtering Pathfinder performs and the limited manual investigation of slices. We only investigate as many slices as needed to understand the relation between threads (e.g. message passing, shared cache, shared application state). On average we investigated 3 slices for one thread pair.

7.2.7 Instrumentation Time

To provide qualitative context to the instrumentation presented here, we report the wall-clock time needed to complete the instrumentation. This is based on a single developer (the author of this thesis) and only serves to provide context.

The full instrumentation process for intra-process context propagation took about 10 hours. The investigation of an irrelevant slice took between 30 and 60 seconds on average. Based on the information that are part of the stack traces, it was usually possible to derive the context of the relation and that it is not relevant.

The investigation of relevant slices, and corresponding code places plus the instrumentation took about 3 hours in total. The most time consuming part here was to assess and verify the effects of possible instrumentation places. In Cassandra all execution services share a few common base classes. Pathfinder reveals them but we had to verify that the general instrumentation of a base class does not affect branches where propagation is undesired (e.g. a long running task is started).

The investigation of seemingly relevant slices and corresponding methods that are eventually not part of the reference instrumentation took about 1 hour.

The rest of the time was spent on preparation, compilation, running the system and loading, filtering, selecting the tracked data.

7.2.8 Search Space

To give an indication that Pathfinder indeed reaches a reduction in search space, we relate the code part we covered while the instrumentation with Pathfinder to Cassandra's code base.

The code base of Cassandra includes 1678 Java files, excluding testing and third-party code. These files comprise 4621 classes and 28172 methods.

The communication slices we investigated with Pathfinder Explorer covered 801 methods (3% of the code base). That is, 3% of methods appeared on either the writer or reader stacks traces (the average stack trace had a depth of

Request Type	Iterations	Distinct Slices	Investigated Slices		Covered Classes		Covered Methods		Investigated Methods		Instr. Points Found
			Rel.	Total	Rel.	Total	Rel.	Total	Rel.	Total	
create table	4	4.5k	19	178	53	331	107	769	22	53	5
put	1	0.5k	0	6	0	50	0	87	0	0	0
get	1	0.1k	0	2	0	41	0	64	0	0	0
scan	1	0.5k	0	7	0	63	0	107	0	1	0
delete	1	0.5k	0	5	0	44	0	79	0	0	0
disable table	2	1.5k	3	26	12	125	20	230	3	3	1
delete table	1	2.0k	0	6	0	26	0	50	0	0	0
enable table	2	1.5k	0	37	0	135	0	268	0	3	0
System	10	11.1k	22	219	58	381	114	918	25	60	6

(a) Summary statistics for instrumenting HBase with Pathfinder. Request types were instrumented in the order presented. Slices, classes, and methods are *relevant* if they correspond to some instrumentation in the baseline manual instrumentation.

Request Type	Iterations	Distinct Slices	Investigated Slices		Covered Classes		Covered Methods		Investigated Methods		Instr. Points Found
			Rel.	Total	Rel.	Total	Rel.	Total	Rel.	Total	
create table	–	–	+14	–	+23	+18	+67	+42	+13	–	+5
enable table	–	–	+3	–	+11	+11	+18	+13	+3	–	+1
System	–	–	+17	–	+31	+26	+72	+45	+16	–	+6

(b) Using Pathfinder we identified several places that were *missing* from the baseline instrumentation. These numbers describe the additional slices, classes, and methods to find these additional instrumentation points. They occurred in two request types.

Table 7.2: Summary statistics for instrumenting HBase with Pathfinder. See §7.3.2 for a detailed description.

20 methods). As part of the high-level investigation of slices we scanned parts of the stack traces to understand the communication between threads.

During our investigation, we investigated the code of 46 methods in detail. These are part of 34 classes. 46 methods correspond to 0.16% of the code base.

7.3 Instrumenting HBase

In this section we repeat the experiment presented in the previous section, this time considering HBase [25]. HBase is an open-source distributed database that interacts with two other services, ZooKeeper [61] and HDFS [56]. It is widely used as part of larger microservice architectures.

7.3.1 Setup

We use HBase version 1.6 [25] with the extended instrumentation patch applied [66]. We ran a 2 node cluster with one master node and one region node. They all run as separate processes on the same host. Similar to Cassandra this small setup is sufficient to exercise all code paths.

We modified the source code of HBase to prepare the application of Pathfinder, by adding inter-process instrumentation (`serialize` and `deserialize`) in 4 locations in code. This enables Pathfinder to track slices between processes. As with Cassandra, identifying these locations is trivial. A request timeout adjustment was not necessary. The default is 300 seconds.

7.3.2 Results

We instrumented 8 request types, shown in Table 7.2a, in the given order. The order is arbitrary, but relevant because instrumentation found for one request type is then already covered for later request types. The table has the same structure as Table 7.1 and we perceive similar results. Only a small number of slices need to be investigated to identify appropriate instrumentation points. There are a larger number of irrelevant slices. And as we proceed through the request types, less additional slices were necessary to investigate because common code locations are already covered by instrumentation of earlier request types.

7.3.3 Derived Instrumentation

Using Pathfinder, all instrumentation points of the reference instrumentation were also identified and instrumented. HBase has more instrumentation points than Cassandra because in Cassandra all request related ad-hoc executions are performed by execution services that share a set of base classes and most of the instrumentation needed was within those base classes. This is *not* the case for HBase. So for HBase we had to put instrumentation close to places where the execution services were *used* (e.g. wrapping a service instance) instead of instrumenting the service classes generally.

In addition to this aspect, HBase is more heterogeneous in how tasks are executed (e.g. different custom execution services, ad-hoc thread creations) and exhibits more infrastructures that are instrumented in the reference instrumentation such as an event handling service, procedure scheduling service and an asynchronous commit log.

As for Cassandra, Pathfinder identified several places in HBase which are not covered by the reference instrumentation and that are not necessarily relevant but optional (e.g. message passing to extra thread that performs an RPC). We did not consider these relevant in Table 7.2.

Unlike Cassandra, we also identified several further places using Pathfinder that are clearly part of the request execution but were missing in the reference instrumentation. The missing instrumentation corresponds to features for which the official instrumentation is still work in progress [67]. So the missing instrumentation points may be added to the official instrumentation in the future. Table 7.2b lists the additional relevant slices, classes, and methods for these locations.

7.3.4 Instrumentation Time

As with our Cassandra instrumentation, we report the wall-clock time needed to complete the instrumentation. This provides qualitative context for the experiment presented here. The full instrumentation process for intra-process context propagation took about 16 hours. The investigation of an irrelevant ITCE took between 30 and 60 seconds on average. Based on the information that are part of the stack traces, it was usually possible to derive the context of the relation and that it is not relevant. The investigation of relevant slices, and corresponding code places plus the instrumentation took about 6 hours in total. The investigation of seemingly relevant slices and corresponding methods that are eventually not part of the reference instrumentation took about 2 hours. The rest of the time was spent on preparation, compilation, running the system and loading/filtering/selecting the tracked data.

7.3.5 Search Space

As for Cassandra we relate the code part we covered while the instrumentation with Pathfinder to HBase's code base.

The code base of HBase includes 1919 Java files that do not contain testing or third-party code. These include 3427 classes and 26072 methods. With the help of Pathfinder we investigated the code of 60 methods that are part of 42 classes. 60 methods correspond to 0.2% of the code base. The high level investigation of inter-thread slices covered 918 methods what is 4% of the code base.

Only 1900 HBase specific classes were actually loaded by HBase at runtime to execute the listed request types. This number does *not* include generated classes, Java library classes or other third-party libraries HBase uses. If we assume a developer is able to identify those classes at no cost as part of a code traversal when instrumenting manually we can use this set of classes as resulting code base. In comparison to this class set we investigated 2% of the classes.

7.3.6 Additional Insights

- *Detecting inter-process boundaries* – Although Pathfinder does not explicitly support the instrumentation of *process* boundaries, Pathfinder was able to identify all inter-process message sending points in Cassandra and many in HBase. This was possible because most used RPC frameworks use separate threads to perform the RPC. Pathfinder detects communication to those threads and captures the accesses to message objects when those are created and serialized by different threads.
- *Detecting system level transactions* – Pathfinder was able to identify several system internal operations and their start points in code. System internal operations and background tasks are started as part of request executions or access resources that were written by request executions. So the threads that execute those operations appear as readers of read-after-write relations Pathfinder tracks. The reported stack traces point to corresponding methods and operation start points. In Cassandra those operations are not traced; some of them are traced in

HBase. For the presented instrumentation task we focus on user-level requests but one could use the identified start points of those operations to also instrument them with Pathfinder.

- *Instrumentation points are shared* – The reference instrumentation of HBase traces system internal transactions which are not triggered by a user (e.g. periodic flushing, pre-fetching). We do not investigate such transactions using Pathfinder, although Pathfinder would be suitable for instrumenting them given some appropriate starting point in code. Nonetheless, we observed that Pathfinder was able to detect all instrumentation points relevant to these background tasks, because different types of requests share the same infrastructures which need to be instrumented. So investigating a subset of requests already allows a complete system instrumentation.
- *Request Profile* – The successive investigation of code slices as part of Pathfinder's workflow automatically leads to an understanding of the execution profile of requests. This allows to asses the context and relevance of potential instrumentation points, shows how they belong together and gives additional confidence that the instrumentation is end-to-end.

7.3.7 Manual Instrumentation Experience

As part of other research projects we instrumented several systems manually and found ourselves often in situations where we could identify several instrumentation points but finding the remaining ones was extremely time consuming or points were completely missed. When the critical end-to-end path was instrumented there was no indication that there are still sub-tasks connected, resulting in an incomplete instrumentation. We experienced a heuristic code traversal as extremely time-consuming, covering an often insufficient sub set of the code base. Which branches to follow is a often an ad-hoc decision and not exhaustive. So the variance in needed time to identify an instrumentation point was usually very high, resulting in instrumentation times up to several weeks.

With Pathfinder the instrumentation process became a straightforward processing of displayed slices. The mental load reduced significantly and presented slices cover an sufficient sub set of code to identify all instrumentation points.

7.3.8 Summary

Overall we show that Pathfinder was able to detect *all* instrumentation points and so that Pathfinder successfully exploits inter-thread communication events to reveal instrumentation points for intra-process context propagation.

We give several indications that Pathfinder achieves a significant reduction in search space and so needed time to accomplish a full system instrumentation.

7.4 How does Pathfinder compare to an automatic instrumentation approach?

In §3.3 we outlined several prior approaches to partially automating instrumentation. In this section we evaluate one such automatic instrumentation approach and compare the resulting instrumentation with the one derived with Pathfinder. Concretely, we use the automatic instrumentation feature of OpenTracing [68]. The experiment illustrates that an existing automatic approach misses many important instrumentation points, but also adds instrumentation to many places where it is undesirable.

7.4.1 OpenTracing Instrumentation Agent

OpenTracing provides an automatic code transformation tool that plugs-in pre-instrumented libraries and instruments fixed code patterns. Classes are modified at runtime with byte code instrumentation. The application of this tool only requires to provide OpenTracing's agent to the Java virtual machine of the target program.

In the context of intra-process context propagation in HBase and Cassandra the agent instruments classes that inherit or implement the following classes/interfaces.

- `java.util.concurrent.Executor`
- `java.util.concurrent.ScheduledExecutorService`
- `java.lang.Thread`

`Executor` and `ScheduledExecutorService` are execution infrastructures that are driven by thread pools and allow to submit tasks which are then executed by a thread pool thread. The instrumentation the agent inserts for

`java.util.concurrent.Executor` and `java.util.concurrent.ScheduledExecutorService` propagates context along with the submission of task objects and for `java.lang.Threads` the context is propagated on thread creation and continued when the thread is started. The agent does not help finding start points of requests just as Pathfinder.

7.4.2 Results

Cassandra

We used the OpenTracing agent to instrument Cassandra. The resulting instrumentation is over-complete: it covers all instrumentation points of the reference instrumentation, but also many others that do not require instrumentation. The agent instruments 43 classes for intra-process context propagation where only 2 are actually instrumented in the reference instrumentation.

The instrumentation is incorrect or unsuitable at many places, resulting in:

- Background tasks that are connected to requests that initially schedule them.
- Infinite traces for long running and periodic tasks.
- Connection of low level tasks with a request that the reference instrumentation does not connect to not overload traces.

The tool is fully automatic and so the application did not require any additional effort unlike Pathfinder.

HBase

We could not apply the agent on HBase. HBase crashed while startup when using the agent. We tried to simulate the effects and look for classes that implement the interfaces the agent would look for. Under this simulated application the agent would be able to instrument 4/6 respectively 7/10 (extended baseline instrumentation) instrumentation points the reference instrumentation covers. The missing instrumentation points appear at custom services which do not use any Java execution infrastructures. At such places item processing is handled by threads which are manually started and stopped and items are passed over generic data structures such as queues. Request context has to be propagated when requests submit items to those services. The agent cannot handle such scenarios.

As for Cassandra the agent generally instruments all classes that inherit from Java execution infrastructure classes. This results in a largely over-complete instrumentation. We did not simulate the full extend of the agent's automatic instrumentation but we found 95 classes that implement `java.util.concurrent.Executor` and identified over 5 long running threads and 4 tasks that perform periodic executions. The instrumentation at these places leads to similar effects like mentioned for Cassandra (e.g. infinite traces, connection of background tasks with the requests that initially schedule them)

7.5 Overhead

In the following we present measurements of overhead Pathfinder introduces for request executions. The overhead Pathfinder introduces for request executions must be sufficiently low such that requests complete successfully, do not time out or are distorted. Otherwise Pathfinder is not able to capture all the slices and inter-thread communication channels a request usually covers.

The only performance requirement for Pathfinder is that requests correctly execute. Pathfinder is exclusively applied at development time and does not affect executions at production time. The final instrumentation for a system that runs in production is fully independent of Pathfinder.

We measure runtimes for normal request executions and compare them with tracked request executions under Pathfinder. Additionally, we check if the requests complete successfully and if any problematic events are logged while the execution.

The results show that Pathfinder introduces significant overhead for requests. Request executions under Pathfinder can be over 150 slower than normal. However, with appropriate timeout adjustments all requests complete successfully and do not show any odd behavior. We list possible optimizations to improve the performance of Pathfinder in §8.2.

7.5.1 General Setup

The basic setup is the same as presented in §7.2.1 and §7.3.1. In addition we consider the widest end-to-end tracking setting the systems had while the instrumentation process. This is the worst case scenario where all appearing slices are tracked along the end-to-end path of a request. In addition, we limit the slices that Pathfinder tracks per thread pair to 50. Pathfinder ignores further appearing slices. We also used this setting when doing the instrumentation in §7.1.3.

All executed requests insert, delete or retrieve single values. We never run data heavy workloads.

7.5.2 Single Execution

In this experiment we measure the overhead of tracked requests when Pathfinder tracks exactly one request right after system start. This scenario is the same as when we did the instrumentation in §7.1.3.
We compare the runtime of a normal request execution to the runtime of a tracked request execution.

Before we measure the runtime of the tracked request execution we run a small workload that contains a single execution of all the requests that we want to instrument. This setup mimics the Pathfinder application scenario we presented in §7.1.3. In each iteration when Pathfinder is applied, one compiles the system, runs it, executes a small workload before tracking exactly one request. The tracked request execution has to complete successfully such that Pathfinder can be effective.

Results

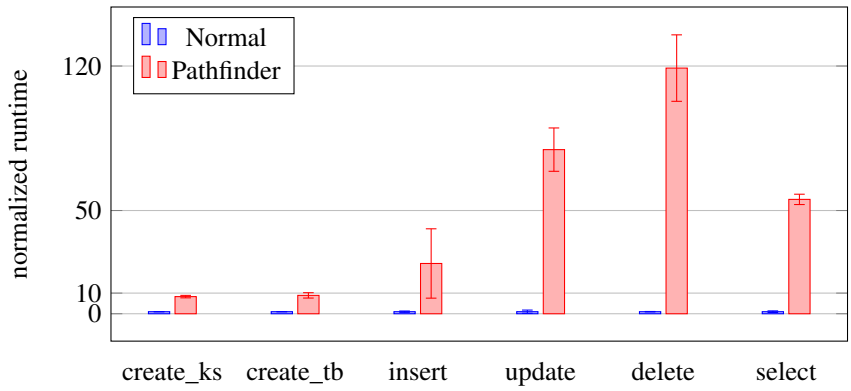

Figure 7.1: **Cassandra**: Normalized runtimes of the requests under normal operation and under Pathfinder. Each request execution was the first after running a small workload after system start. Average and std.-dev. of 5 first request executions directly after system start.

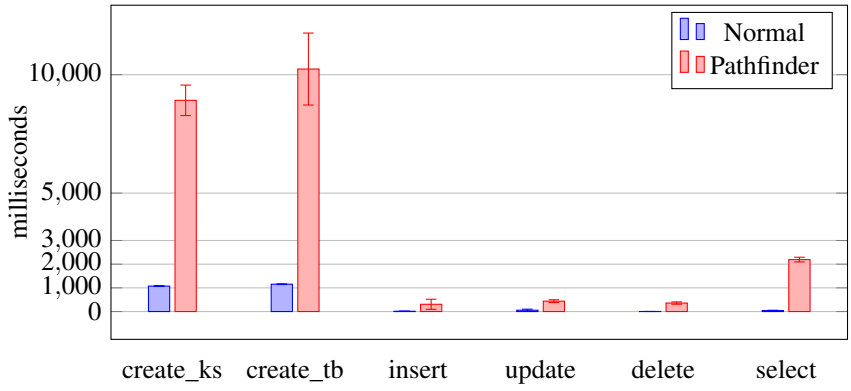

Figure 7.2: **Cassandra**: Absolute runtimes of requests under normal operation and under Pathfinder. Each request execution was the first after system start. Average and std.-dev. of 5 first request executions directly after system start.

Figure 7.1 and Figure 7.3 show normalized average (μ) runtimes for Cassandra respectively HBase. The line on each bar covers the range from $[\mu - \sigma, \mu + \sigma]$. We averaged 5 first executions.

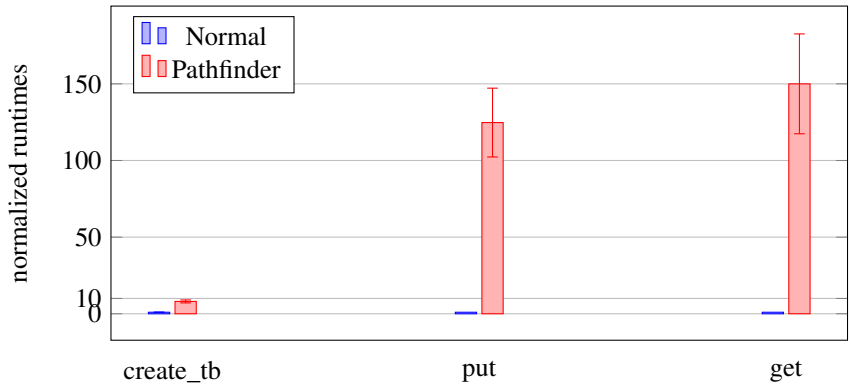

Figure 7.3: **HBASE**: Normalized runtimes of the requests under normal operation and under Pathfinder. Each request execution was the first after system start. Average and std.-dev. of 5 first request executions directly after system start.

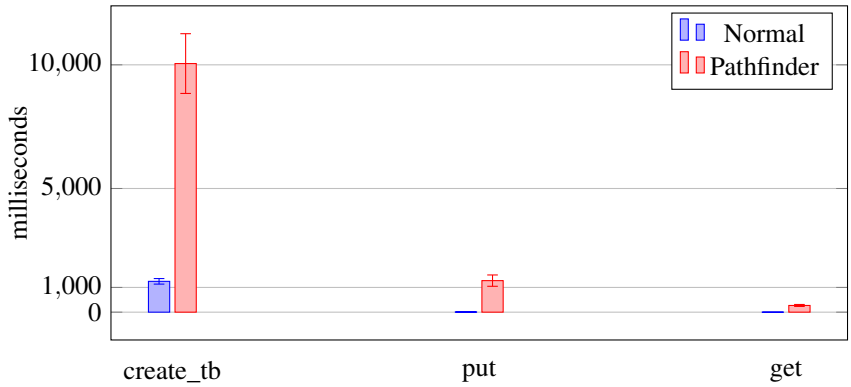

Figure 7.4: **HBASE**: Absolute runtimes of the requests under normal operation and under Pathfinder. Each request execution was the first after system start. Average and std.-dev. of 5 first request executions directly after system start.

Figure 7.2 and Figure 7.4 show the corresponding absolute runtimes. Normal runtimes for delete, insert, update, select, put and get lie between 1 and 60 ms.

The overheads are significant for all request types. The first execution of requests can be over 150 times slower than normal. This is because Pathfinder captures every memory read; and every memory write under a tracking scope. Every memory write under a tracking scope and every memory read that leads to an ITCE causes the capturing of the reader's stack traces and leads to a log entry that is hold in memory.

The absolute overhead of Pathfinder is higher for operations that effect more state like `create table` and `create keyspace` that cause more ITCEs. This is also visible in Table 7.1 and Table 7.2 where we see more tracked slices for the operations for which Pathfinder causes more absolute runtime overhead.
However, the relative overhead is higher for fast operations like `get` and `put` and lower for operations like `create table`. A possible explanation for this is that a significantly bigger portion of the runtime for `create table` is caused by RPCs and disk accesses in comparison to runtimes of `put/get`. Pathfinder does not cause overhead for message delivery over sockets or disk accesses. So these operations contribute to the overall runtime but do not increase the absolute overhead Pathfinder causes. So the overall relative overhead Pathfinder causes appears lower.

For `create keyspace`, `create table` and `select` requests we have to increase the timeout (default is 2 seconds) such that requests do not abort. With the adjusted timeout all requests complete successfully and do not exhibit any distorted behavior visible in the logs.

Overall we see that with a timeout adjustment Cassandra tolerates the overhead and requests execute correctly and complete successfully.

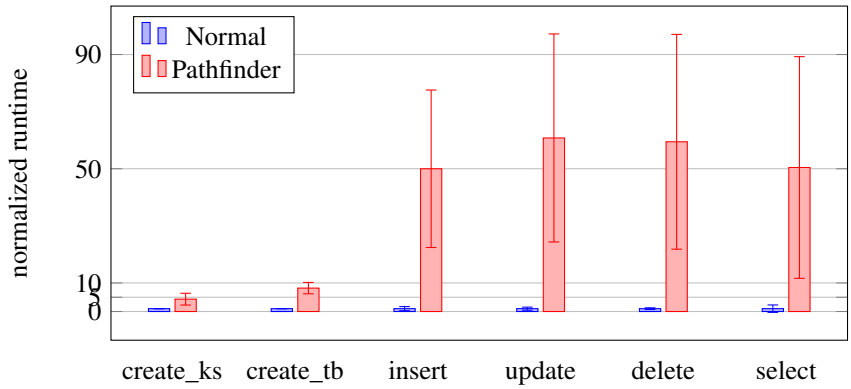

Figure 7.5: **Cassandra**: Normalized runtimes of the requests under normal operation and under Pathfinder. Average and std.-dev. of 25 request executions directly after system start.

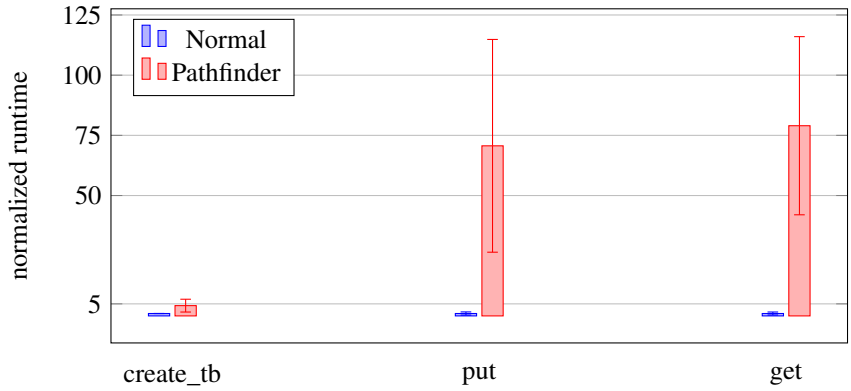

Figure 7.6: **HBASE**: Normalized runtimes of the requests under normal operation and under Pathfinder. Average and std.-dev. of 25 request executions directly after system start.

7.5.3 Multi Execution

In this experiment we measure the overhead of tracked requests when Pathfinder tracks multiple executions of requests right after system start. This experiment is similar to the one presented in §7.5.2. However, now we run requests multiple times instead of exactly once such as in a small workload. This scenario can be relevant when tracking cross request relations (e.g. shared caches, join points). The main question this experiment wants to answer is, if the overhead for requests increases with an increasing amount of executions.
We compare the runtime of a normal request execution to the runtime of a tracked request execution.
Before we measure the runtime of the tracked request execution we run a small workload as in experiment §7.5.2.

Results

Each measurement shown in Figure 7.5 and Figure 7.6 corresponds to the average of 25 executions of one request after executing a small workload right after the system start.

The results show that the overhead tends to decrease with more executions for within 25 executions. This is because the first tracking activity for the first execution leads to initialization of Pathfinder's tracking framework and Java's just-in-time compiler can bring in optimizations over time. With more and more tracking activity Pathfinder accumulates more and more data that is considered for lookups when a memory address is accessed for reading by a thread. The results show that this accumulation does not significantly increase the runtime for such lookups when small workloads are tracked.

Chapter 8

Discussion

8.1 Approximating Developer Effort

In our evaluation, we measured several statistics relating to the system code base, to illustrate that Pathfinder reduces the amount of code a developer must consider. These statistics are only approximate measures that, we argue, correlate with developer instrumentation effort. However, we do not compare to the amount of code that is actually investigated as part of a manual instrumentation process.

To address this, we could conduct a user study where a group of users instruments a set of systems with Pathfinder and another instruments the same systems without Pathfinder. The needed time and instrumentation experience could be measured and compared. We did not conduct a user study as it was beyond the scope of this work. Moreover, instrumentation is a specialized task, for which it is challenging to identify experts to perform the task. Further, the needed time each user has to invest to instrument the systems can take days to weeks. Setting such a user study up would be far too costly and time intensive as part of this thesis. The focus of the presented evaluation is on measuring how many instrumentation points Pathfinder is able to detect, showing that observing inter-thread communication events can be effectively exploited to identify context propagation places in real systems. We reported on the wall-clock time taken to perform this instrumentation by the author of this thesis. Anecdotally, this instrumentation process was substantially faster than manual instrumentation tasks performed by the same author previously.

We do not evaluate Pathfinder on other uses case than distributed tracing, since we could not find any open source systems that have a corresponding instrumentation in place. Defining a reference instrumentation on our own would be biased and potentially unrealistic. Our evaluation approach fundamentally relies on a realistic reference instrumentation.

8.2 Pathfinder's Overhead

In §7.5 we showed that Pathfinder causes significant overhead for tracked requests. Since Pathfinder is exclusively applied at development time, this is not a problem as long as timeouts can be adjusted accordingly. There can be many factors in realistic scenarios which can naturally cause a massive latency increase for requests in distributed systems. A system that is not configurable to tolerate such higher latencies and just does not properly serve requests in such common scenarios seems unlikely.

- *RPC distance* – Network messages might be transferred between data-centers across continents. What causes high message delivery times (can be order of seconds).
- *RPC loss/delay* – Networks are best effort. Packages can get lost or delay causing high request latencies.
- *System under load* – If a services gets popular request amounts can suddenly increase. The available resource per request decreases and request latencies increase.
- *Slow nodes* – Single nodes that are traversed by a request can be exceptionally slow because of high disk failure rates.
- *Heterogeneous machine power* – The same system might be deployed for different applications under strongly different resource conditions what causes higher or lower request latencies and request latency requirements.

However, we cannot exclude the possibility that systems exist that do not tolerate the overhead Pathfinder introduces. We think Pathfinder's tracking performance can be significantly improved to increase the probability that systems

tolerate the caused overhead. We identified several places in our prototype where optimizations are possible to improve performance.

- *Initialization* – Measurements presented in §7.5.3 give an indication that Pathfinder's initialization causes significant overhead. Pathfinder is initialized when the tracking framework is first touched by the first tracked request. To decrease overhead for the request, we could decouple the initialization. We could initialize Pathfinder with a separate initialization request that just initializes pathfinder on all nodes before an actual request is tracked. We do not initialize Pathfinder statically when a node is started to prevent class loader deadlocks.
- *Synchronization* – Pathfinder globally serializes tracking calls for simplicity. There is no interleaving between calls to Pathfinder's tracking component that happen by different threads. This is not necessary. Only the access to the central look up map must be synchronized properly. This can lead to waiting times for a thread when the tracking component is currently blocked by another. Since all reads from shared memory are redirected to Pathfinder's tracking framework, the overall wait times of all threads might be significant.
- *Stack traces* – Pathfinder frequently captures stack traces and capturing a stack trace is expensive in Java. Improving the stack trace capturing time will improve the track latency. This can be achieved by actively maintaining stack traces in a thread's local storage as a thread enters and leaves methods. Capturing a stack trace becomes passing a pointer to the current stack trace state.
- *Data Locality* – Pathfinder maintains one central map to lookup past writes to memory locations. Accesses on this map must be serialized and tracked data is stored not local to the actual memory addresses that are observed. Instead of storing metadata centrally, it is possible to store it locally to the memory address we track accesses to. This would significantly increase spatial locality (cache efficiency) and would remove the needed synchronization of threads.

8.3 Can Pathfinder miss something?

Pathfinder fundamentally relies on detecting inter-thread communication events. However, theoretically there can exist execution patterns which are single threaded and still need context propagation. An example is when only one thread processes and interleaves incoming requests. We presented this example in §2.6. Pathfinder is at its current state not able to detect corresponding propagation places, because there is no inter-thread communication involved. However, our evaluation indicates that this execution pattern is uncommon in modern systems that try to maximize concurrency to fully utilize machines.

In the following we present an example that we encountered in HBase that comes at least close to the execution pattern described above. In HBase we encountered a custom procedure execution stage where procedures are resubmitted. A procedure processing thread pulls a procedure from a queue, processes parts of it, and then resubmits the procedure to the queue. After processing other procedures the thread pulls and continues the procedure.
Pathfinder could detect the re-submission point because the stage is driven by several threads and another thread continued the execution. However, this additional slice was not even necessary. Pathfinder detects the communication between the external thread that initially submitted a procedure and the worker thread that processes it. The general submission and retrieving logic was revealed by the corresponding slices. The derived instrumentation was naturally on a general level such that it covered the re-submission case. We did not specifically addressed this case in any way. So even if the stage had been driven by a single thread we still would have covered this place. We think other possibly single threaded stages like event engines, request dispatching engines can be handled similarly. Revealing the first interaction with the stage is often sufficient.

8.3.1 Adjustment

Even if there does not seem to be necessary, there is a simple adjustment for Pathfinder to extend the tracking capability to reveal such places where context propagation is needed but no inter-thread communication is involved. Instead of associating thread IDs with memory addresses, we associate request IDs with memory addresses. A thread gets a request ID at the start of a request when a tracking scope is begun. When the tracking scope context is propagated, the request ID follows. When a thread writes to a memory address, the request ID is associated with the memory address. Threads that execute code outside of tracking scopes automatically carry a "no-request ID" and if a thread with a "no-request ID" reads from a memory address that has a request ID, a transfer point (similar to ITCE) is detected.
Suppose in the example we described above only one thread would pull, process and resubmit procedures. The thread resubmits the procedure under an active tracking scope and discards the tracking scope afterwards when it ends the current processing. When the thread then pulls the procedure again later the thread pulls the procedure outside of a tracking scope and so the read is performed by a thread with a "no-request ID" - a request ID that differs from the

request ID that was associated with the memory address. A transfer point is detected. Capturing a slice including writer and reader stack traces would reveal the needed instrumentation point.
Scenarios we encountered in the wild that are potentially single threaded like event engines or request interleaving can be revealed this way. However, this adjustment has a few weaknesses.

- Potentially many places will be detected as transfer points which are irrelevant. Whenever a thread accesses memory addresses outside of a tracking scope, it has written before under a tracking scope, these will be detected as transfer points. This includes all data structures and fields that are used to manage a stage (e.g. counter, list of threads, size field of work item list) and more.
- Theoretically there can be places that are still missed. For example, when a single thread can jump between stages and the producer and consumer point are both covered under the same tracking scope and so nothing is detected.

We leave it to future work to further explore this approach.

Chapter 9

Conclusion

Pathfinder is a developer-in-the-loop tool that guides the developer through the end-to-end instrumentation process for request flow tracking. Fundamentally different to other tools, Pathfinder integrates a developer in the instrumentation process. Pathfinder detects and presents potential instrumentation points to the developer who thereby understands the execution profile of requests and related execution patterns to make system and use case dependent decisions about where and how to propagate context. To the best of our knowledge, Pathfinder is the first tool that interactively assists the developer with the instrumentation task.
The application of Pathfinder on two large distributed systems showed that Pathfinder successfully exploits inter-thread communication events to exhaustively reveal instrumentation points for intra-process context propagation in the area of request flow tracking. We provided several indications that Pathfinder additionally achieves a significant reduction in search space and so needed time to accomplish a full system instrumentation.

Bibliography

[1] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," tech. rep., Google, Inc., 2010.

[2] "Jaeger: open source, end-to-end distributed tracing." `https://www.jaegertracing.io/`. [Online; accessed Nov-2020].

[3] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 589–603, USENIX Association, May 2015.

[4] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger, "Stardust: Tracking activity in a distributed storage system," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2006.

[5] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems," in *25th ACM Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[6] "OpenCensus: Collect metrics and distributed traces." `https://opencensus.io/`. [Online; accessed Nov-2020].

[7] "OpenTelemetry: Capture distributed traces and metrics." `https://opentelemetry.io/`. [Online; accessed Nov-2020].

[8] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, (Hollywood, CA), pp. 107–120, USENIX Association, Oct. 2012.

[9] A. Chanda, A. L. Cox, and W. Zwaenepoel, "Whodunit: Transactional profiling for multi-tier applications," *SIGOPS Oper. Syst. Rev.*, vol. 41, p. 17–30, Mar. 2007.

[10] Y. Yang, L. Wang, J. Gu, and Y. Li, "Transparently capturing request execution path for anomaly detection," *ArXiv*, vol. abs/2001.07276, 2020.

[11] S. Karumuri, "PinTrace: Distributed Tracing at Pinterest." `https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup`, 2016. [Online; accessed Nov-2020].

[12] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[13] "Sigelman, B. H. Towards Turnkey Distributed Tracing." `https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736`, 2016. [Online; accessed Nov-2020].

[14] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–18, 2018.

[15] J. Mace, *A Universal Architecture for Cross-Cutting Tools in Distributed Systems*. PhD thesis, Brown University, 2018.

[16] "ACCUMULO-3507 NamingThreadFactory.newThread should not wrap runnable with TraceRunnable," 2015. `https://issues.apache.org/jira/browse/ACCUMULO-3507`.

[17] "ACCUMULO-4191 Tracing on client can sometimes lose sendMutations events," 2016. `https://issues.apache.org/jira/browse/ACCUMULO-4191`.

[18] "CASSANDRA-7644 Tracing does not log commitlog/memtable ops when the coordinator is a replica," 2014. https://issues.apache.org/jira/browse/CASSANDRA-7644.

[19] "HBASE-13077 BoundedCompletionService doesn't pass trace info to server," 2015. https://issues.apache.org/jira/browse/HBASE-13077.

[20] "HDDS-1151 Propagate the tracing id in ScmBlockLocationProtocol," 2019. https://issues.apache.org/jira/browse/HDDS-1151.

[21] "HDFS-8324 Add trace info to DFSClient#getErasureCodingZoneInfo(..)," 2015. https://issues.apache.org/jira/browse/HDFS-8324.

[22] "Hit the Ground Running with Distributed Tracing Core Concepts from Nike Engineering." https://medium.com/nikeengineering/hit-the-ground-running-with-distributed-tracing-core-concepts-ff5ad47c7058, 2020. [Online; accessed Nov-2020].

[23] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel, "Causeway: Support for controlling and analyzing the execution of multi-tier applications," in *Proceedings of the ACM/IFIP/USENIX 6th International Conference on Middleware*, Middleware'05, (Berlin, Heidelberg), p. 42–59, Springer-Verlag, 2005.

[24] B. Tak, C. Tang, C. Zhang, S. Govindan, and R. C. B. Urgaonkar, "vpath: Precise discovery of request processing paths from black-box observations of thread and network activi-ties," USENIX Annual technical conference, 2009.

[25] Apache Software Foundation, "Hbase." https://hbase.apache.org/. [Online; accessed Nov-2020].

[26] A. S. Foundation, "Cassandra." https://cassandra.apache.org/. [Online; accessed Nov-2020].

[27] Netflix International B.V. , "Netflix." https://www.netflix.com. [Online; accessed Nov-2020].

[28] Google LLC, "YouTube." https://www.youtube.com/. [Online; accessed Nov-2020].

[29] "Conquering Microservices Complexity @Uber with Distributed Tracing." https://www.infoq.com/presentations/uber-microservices-distributed-tracing/, 2019. [Online; accessed Nov-2020].

[30] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "X-trace: A pervasive network tracing framework," in *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.

[31] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic internet services," in *Proceedings International Conference on Dependable Systems and Networks*, pp. 595–604, 2002.

[32] P. Reynolds, J. L. Wiener, J. C. Mogul, M. A. Shah, C. Killian, and A. Vahdat, "Experiences with pip: Finding unexpected behavior in distributed systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (New York, NY, USA), p. 1–2, Association for Computing Machinery, 2005.

[33] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[34] "Lightstep about instrumentation best practices." https://docs.lightstep.com/docs/instrument-your-code. [Online; accessed Nov-2020].

[35] "OpenTracing about instrumenting your application." https://opentracing.io/docs/best-practices/instrumenting-your-application. [Online; accessed Nov-2020].

[36] "Towards Turnkey Distributed Tracing." https://web.archive.org/web/20170116105407/https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736, 2016. [Online; accessed Nov-2020].

[37] "Report about Performance instrumentation for Android apps at Facebook." https://engineering.fb.com/android/performance-instrumentation-for-android-apps/. [Online; accessed Nov-2020].

[38] A. S. Foundation, "Accumulo." https://accumulo.apache.org/. [Online; accessed Nov-2020].

[39] "ACCUMULO-3725: Majc trace tacked onto minc trace.," 2015. https://goo.gl/oee3hF.

[40] "HBASE-15880: RpcClientImpl#tracedWriteRequest incorrectly closes HTrace span," 2017. https://goo.gl/SLBV8Q.

[41] "Datastax-JAVA-815: No tracing results when a RETRY happens.," 2017. https://goo.gl/40AVug.

[42] "Datastax-JAVA-794:Enable tracing accross multiple result pages," 2017. https://goo.gl/lV12lP.

[43] "HBASE-11004: Extend traces through FSHLog#sync.," 2017. https://goo.gl/Y26Ohg.

[44] "CASSANDRA-7657: Tracing doesn't finalize under load when it should.," 2017. https://goo.gl/22Z5Kk.

[45] "HADOOP-13438: Optimize IPC server protobuf decoding.," 2017. https://goo.gl/EdVE5G.

[46] "HADOOP-13473: Tracing in IPC Server is broken.," 2017. https://goo.gl/kms4cN.

[47] "CASSANDRA-10392: Allow Cassandra to trace to custom tracing implementations.," 2017. https://goo.gl/SUy6vA.

[48] "CASSANDRA-11706: Tracing payload not passed through newSession(..).," 2017. https://goo.gl/17n6zC.

[49] "CASSANDRA-12835: Tracing payload not passed from QueryMes- sage to tracing session.," 2017. https://goo.gl/WvNxJs.

[50] "ACCUMULO-1531: file system trace does not work," 2013. https://issues.apache.org/jira/browse/ACCUMULO-1531.

[51] "Discussion about tracing multi-parent scenarios with OpenTracing." https://github.com/opentracing/specification/issues/5#issuecomment-260820014, 2020. [Online; accessed Nov-2020].

[52] "OpenTracing: Vendor-neutral APIs and instrumentation for distributed tracing." https://opentracing.io/. [Online; accessed Nov-2020].

[53] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), p. 85–100, Association for Computing Machinery, 2013.

[54] Y. Jiang, L. R. Sivalingam, S. Nath, and R. Govindan, "Webperf: Evaluating what-if scenarios for cloud-hosted web applications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), p. 258–271, Association for Computing Machinery, 2016.

[55] D. Li, J. Mickens, S. Nath, and L. Ravindranath, "Domino: Understanding wide-area, asynchronous event causality in web applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (New York, NY, USA), p. 182–188, Association for Computing Machinery, 2015.

[56] A. S. Foundation, "Hdfs." https://hadoop.apache.org/docs/r1.2.1/hdfs_user_guide.html. [Online; accessed Nov-2020].

[57] A. S. Foundation, "Hadoop Map Reduce." https://hadoop.apache.org/. [Online; accessed Nov-2020].

[58] A. S. Foundation, "Hadoop YARN." https://hadoop.apache.org/. [Online; accessed Nov-2020].

[59] "Sock Shop." https://github.com/microservices-demo/microservices-demo. [Online; accessed Nov-2020].

[60] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 3–18, Association for Computing Machinery, 2019.

[61] A. S. Foundation, "Zookeeper." https://zookeeper.apache.org/. [Online; accessed Nov-2020].

[62] A. S. Foundation, "Spark." https://spark.apache.org/. [Online; accessed Nov-2020].

[63] "Java Instrumentation Package." https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html. [Online; accessed Nov-2020].

[64] S. Chiba, "Javassist - a reflection-based programming wizard for java," in *International Business Machines Corp*, p. http://www.javassist, 1998.

[65] "Java Native Interface." https://docs.oracle.com/javase/8/docs/technotes/guides/jni/. [Online; accessed Nov-2020].

[66] "Add traces in Procedure V2." https://issues.apache.org/jira/browse/HBASE-23897, 2020. [Online; accessed Nov-2020].

[67] "Replace HTrace with OpenTracing." https://issues.apache.org/jira/browse/HBASE-22120, 2019. [Online; accessed Nov-2020].

[68] "OpenTracing Special Agent." https://github.com/opentracing-contrib/java-specialagent. [Online; accessed Nov-2020].

Appendix A

Visualizations

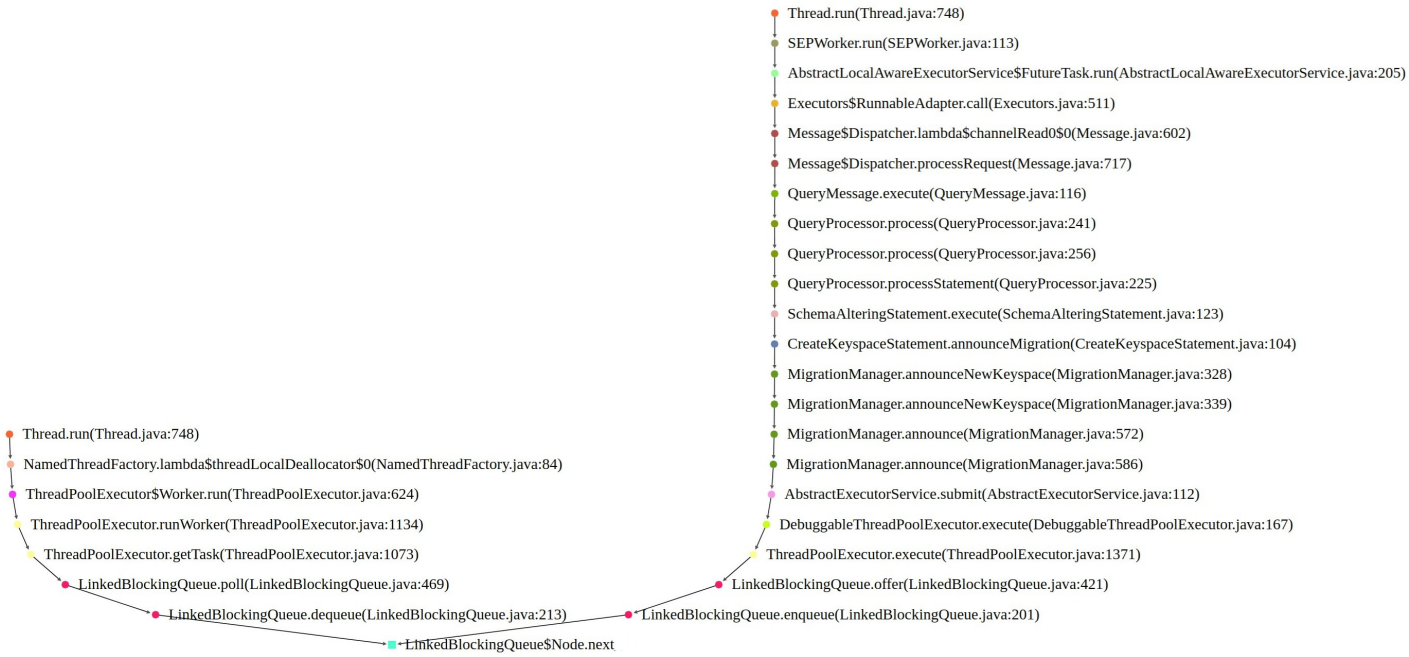

Figure A.1: Cassandra, as part of "Create Keyspace". Writer thread on the right submits work to a thread pool. Reader thread on the left dequeues the work.

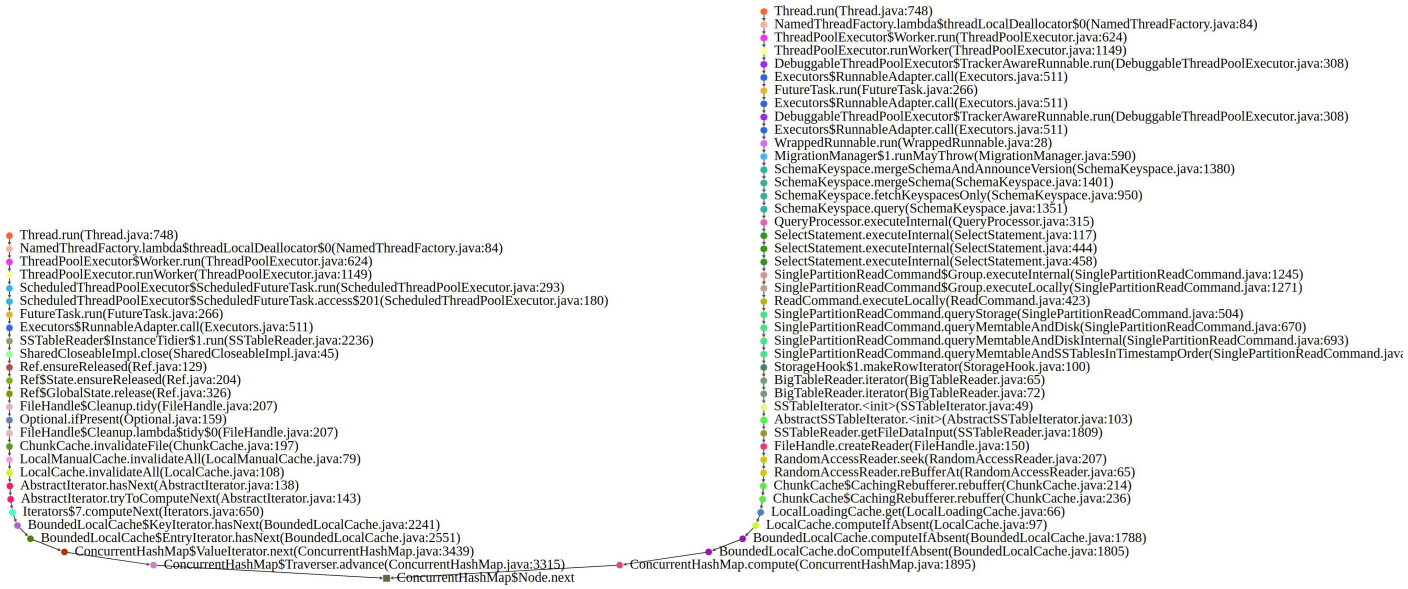

Figure A.2: Cassandra, as part of "Create Keyspace". Writer thread on the right computes cache entry. Reader thread on the left reads from the cache.

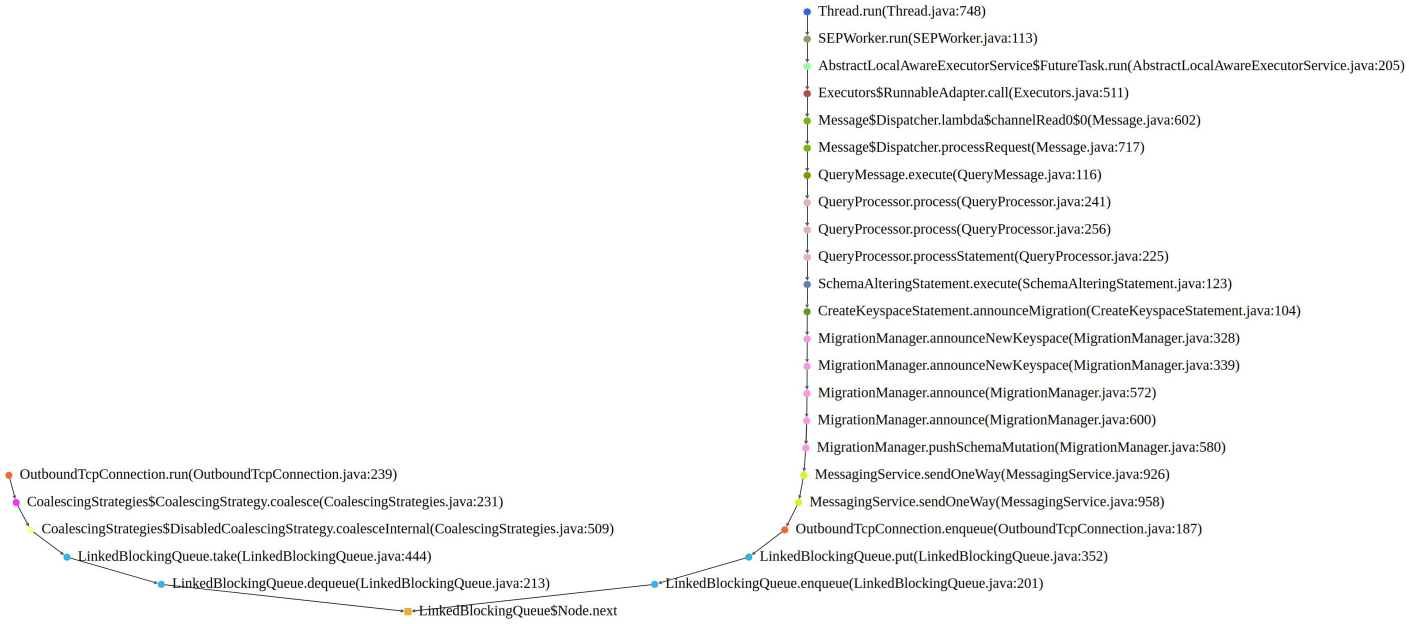

Figure A.3: Cassandra, as part of "Create Keyspace". Writer thread on the right submits message to be send over network. Reader thread on the left pulls the message.

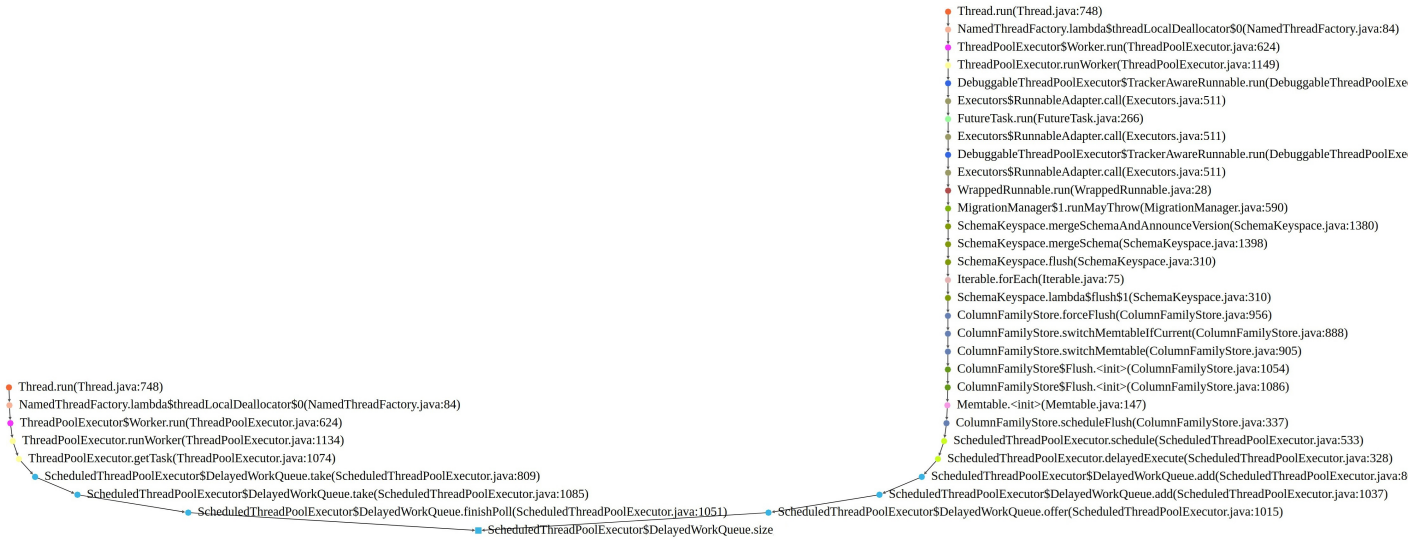

Figure A.4: Cassandra, as part of "Create Keyspace". Memtable flush task scheduling. The writer thread on the right is part of a thread pool and executes a "migration" task that was submitted as part of processing request "create keyspace". On the left the reader thread pulls the scheduled task.

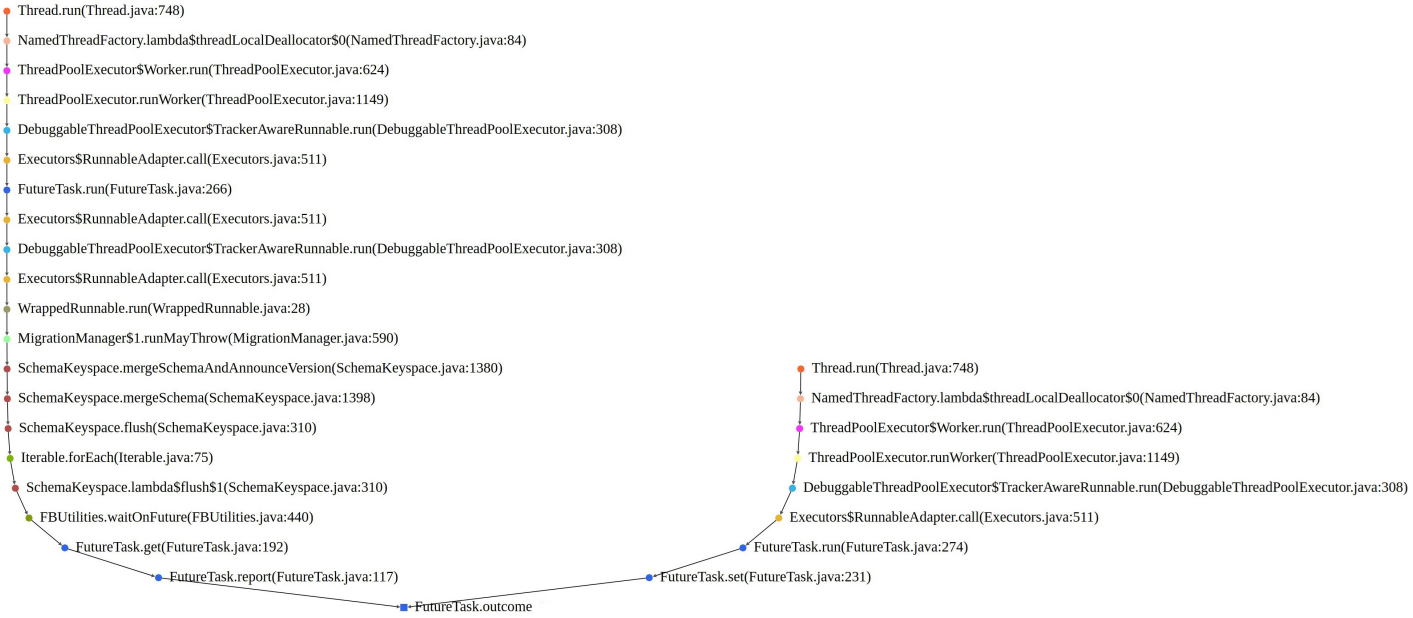

Figure A.5: Cassandra, at several places. Thread pool finished processing and sets result that is read by the reader that initially submitted the task.

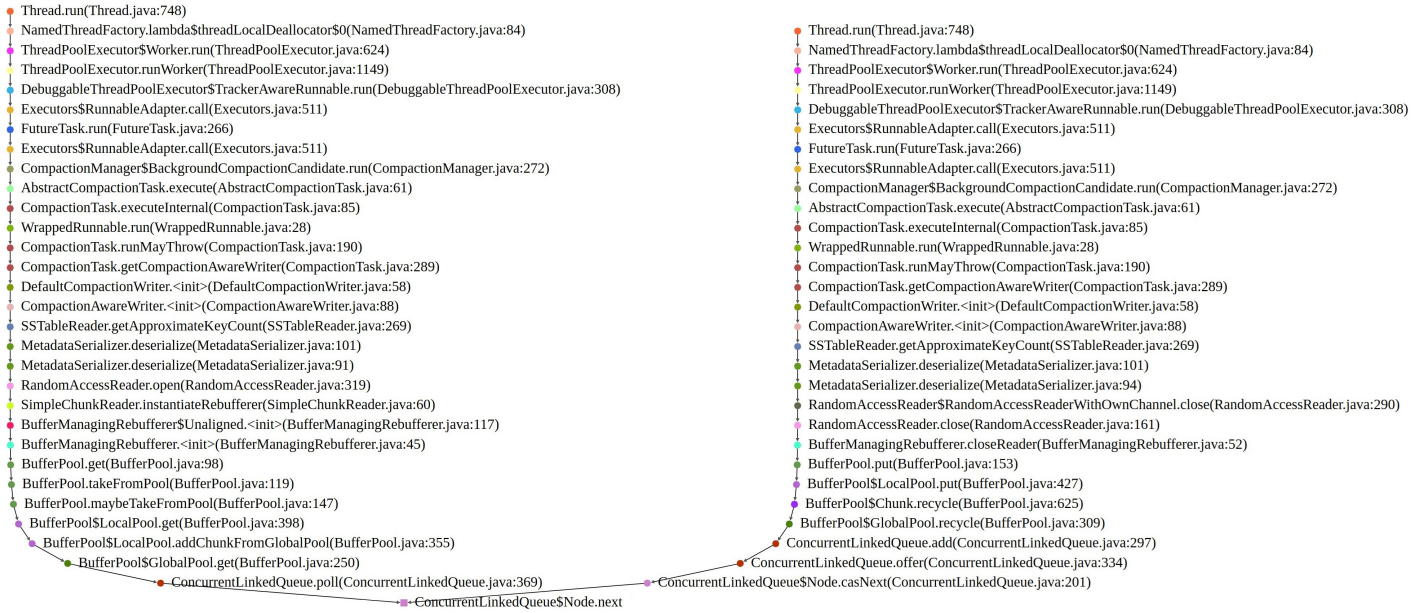

Figure A.6: Cassandra. Writer recycles a buffer and returns it to the buffer pool. Buffer is retrieved by the reader thread.

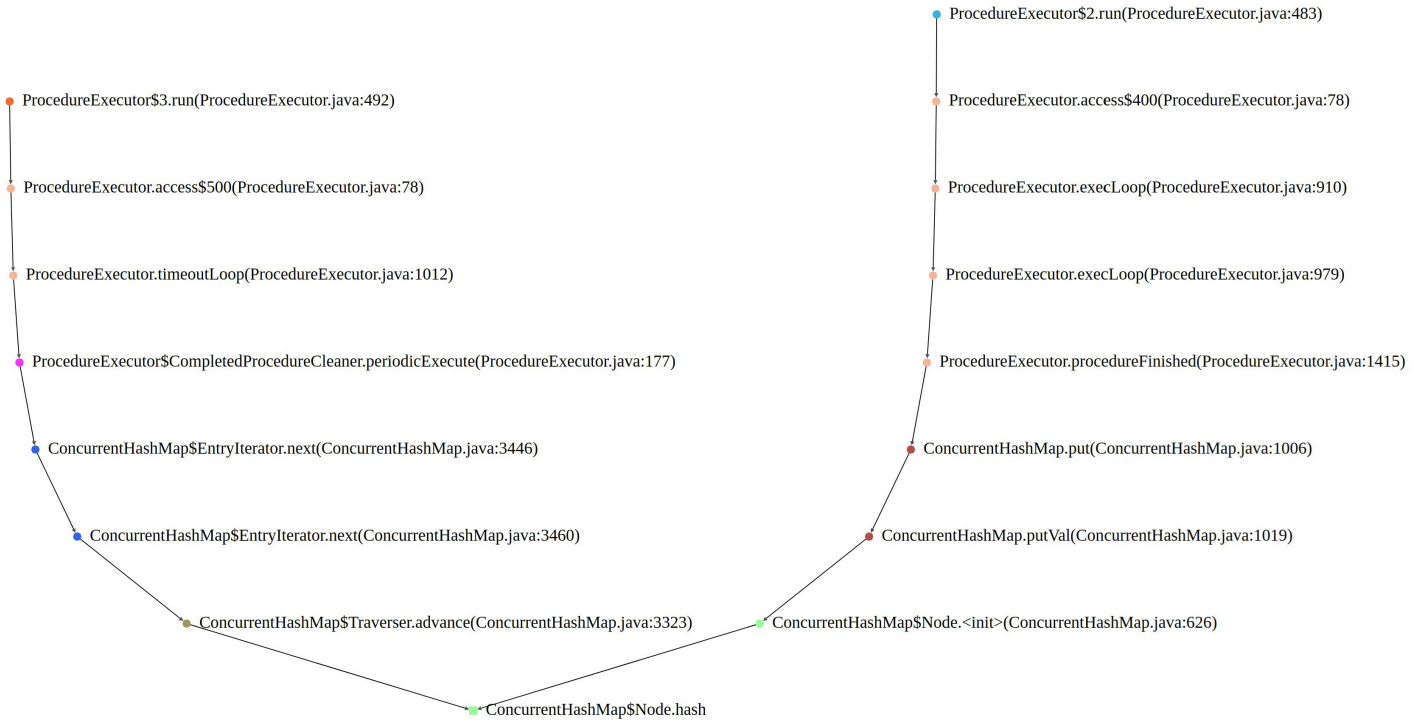

Figure A.7: HBase. The slice reveals a background task. "CompletedProcedureCleaner" iterates over procedures that were marked as finished by the writer thread.

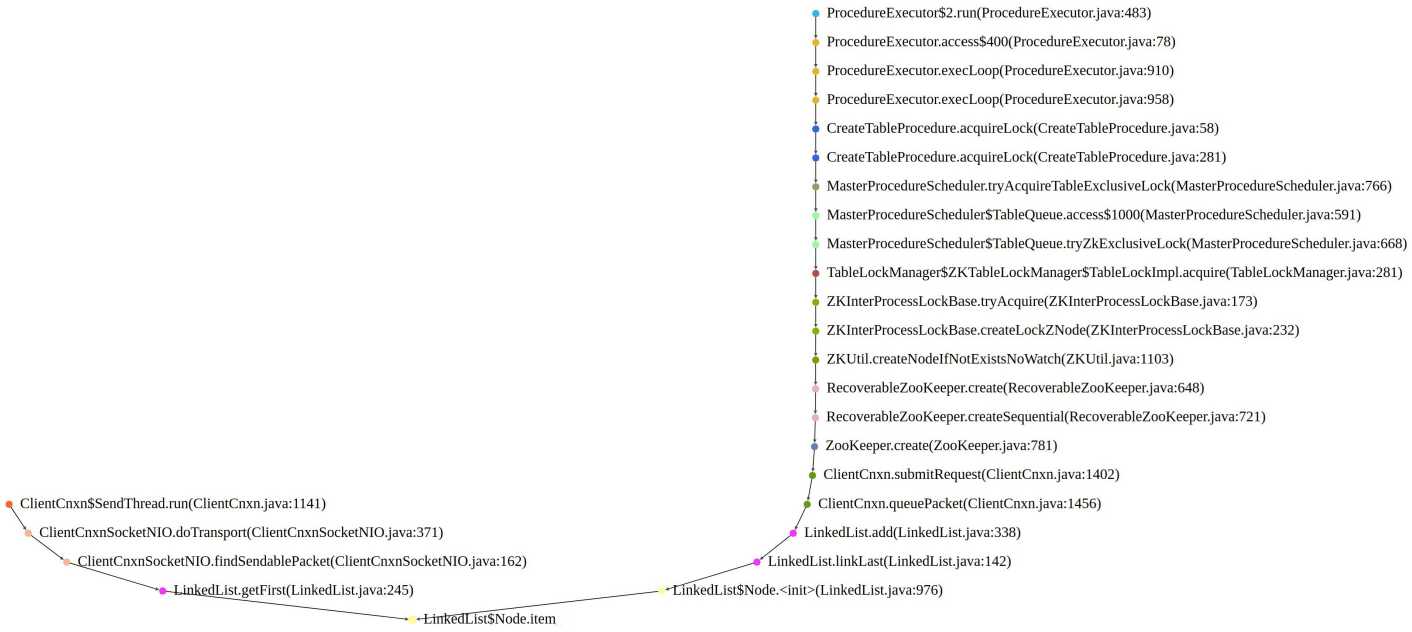

Figure A.8: HBase. Writer thread on the right, passes a message over to a socket thread to send it over the wire.

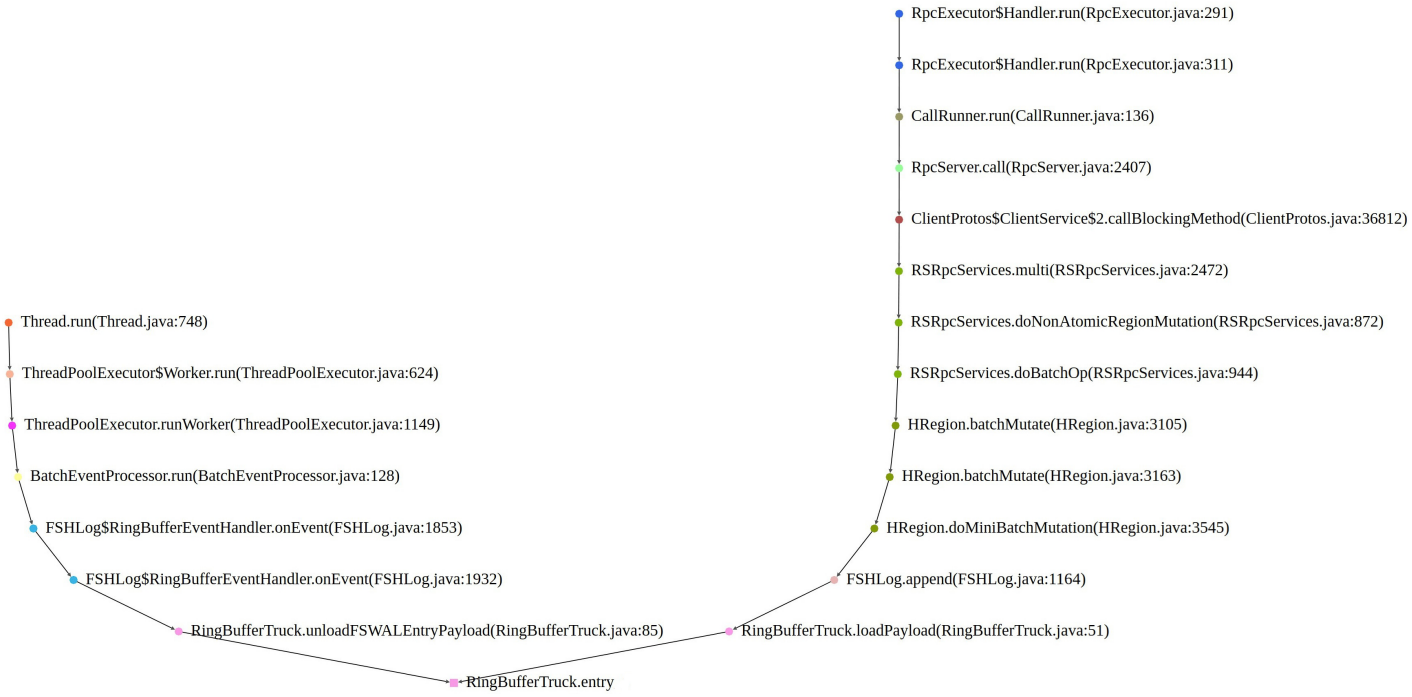

Figure A.9: HBase. Writer thread on the right wants to append a log entry to an operation log. Therefore it submits some log message to an asynchronous operation log service. A thread that is part of this service retrieves the message to perform the append. The message exchange happens via a ring buffer.

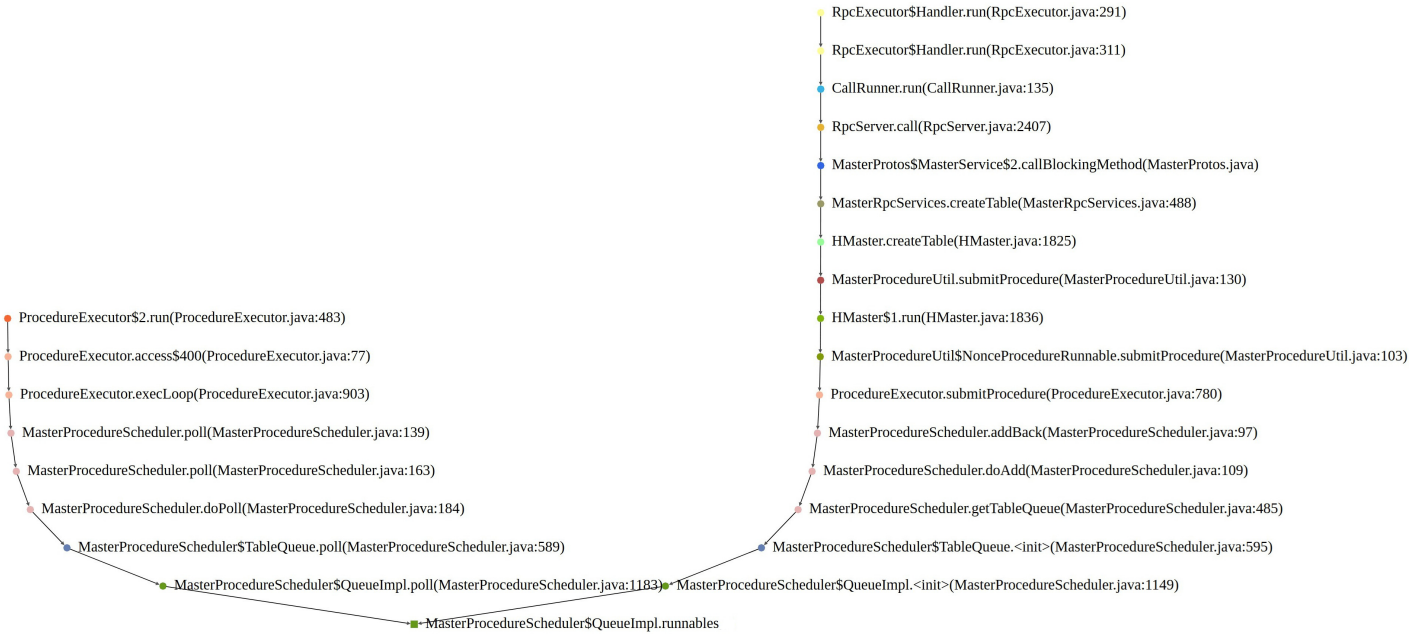

Figure A.10: HBase. As part of processing the "create table" request a procedure is submitted to a procedure scheduler. A procedure executor polls from this scheduler to process the procedure.

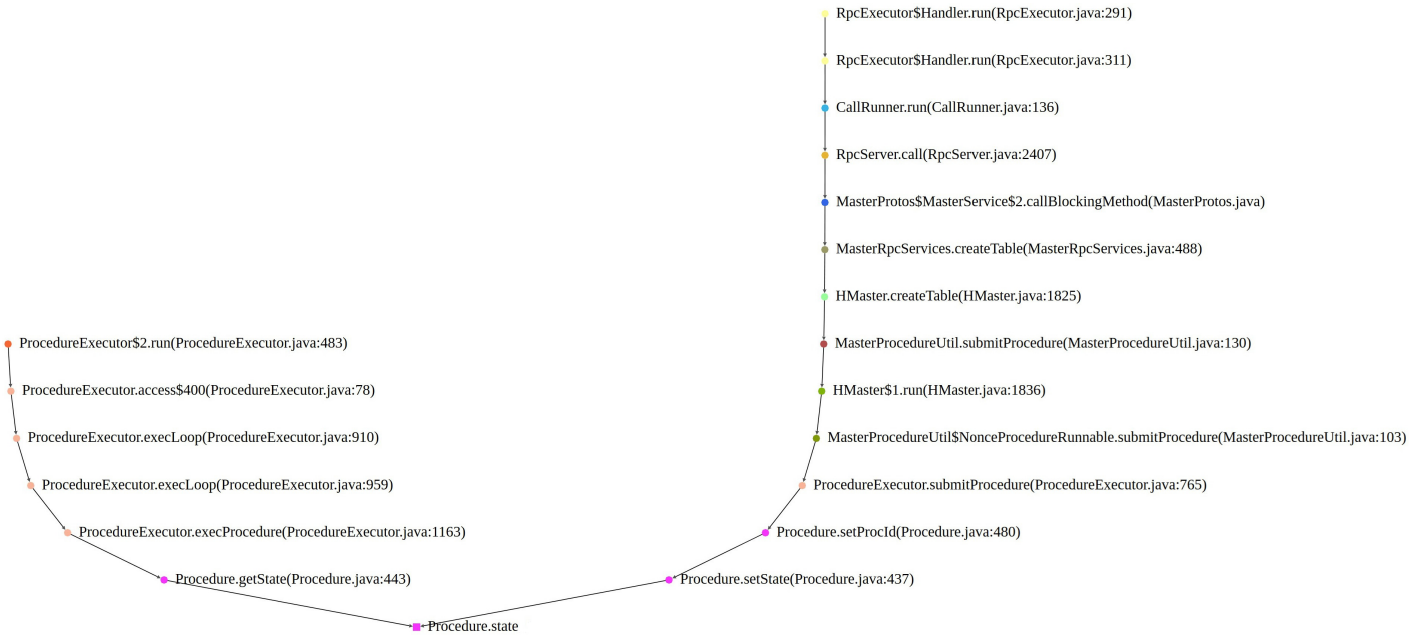

Figure A.11: HBase. This slice appears close by to the slice presented above. As part of processing the "create table" request a procedure is passed to a procedure executor. The "Procedure" object is revealed as message object that is exchanged between the two threads.

Figure A.12: HBase. As part of processing the create table procedure a task is submitted to a thread pool.

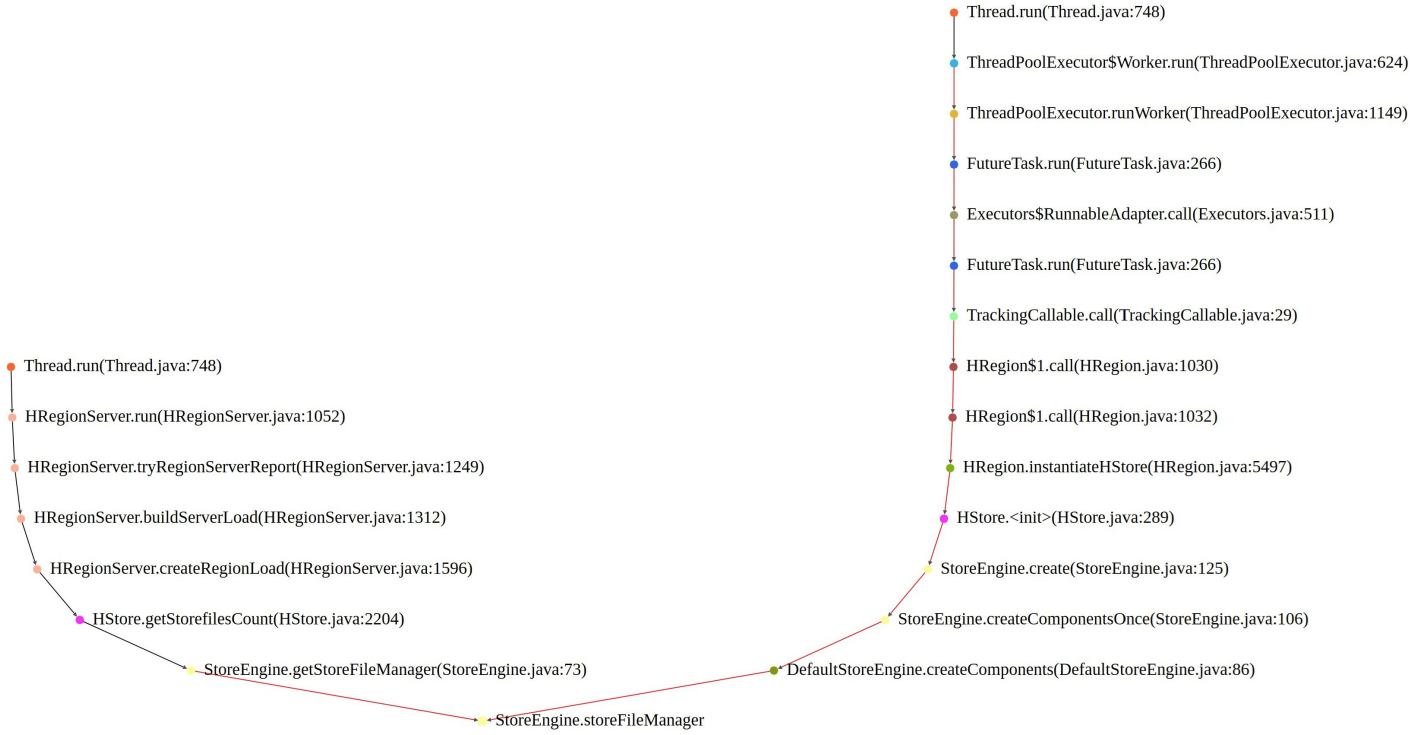

Figure A.13: HBase. The writer thread creates an HStore that is accessed by the reader. The reader thread executes an unrelated request that accesses global application state (i.e. HStore).

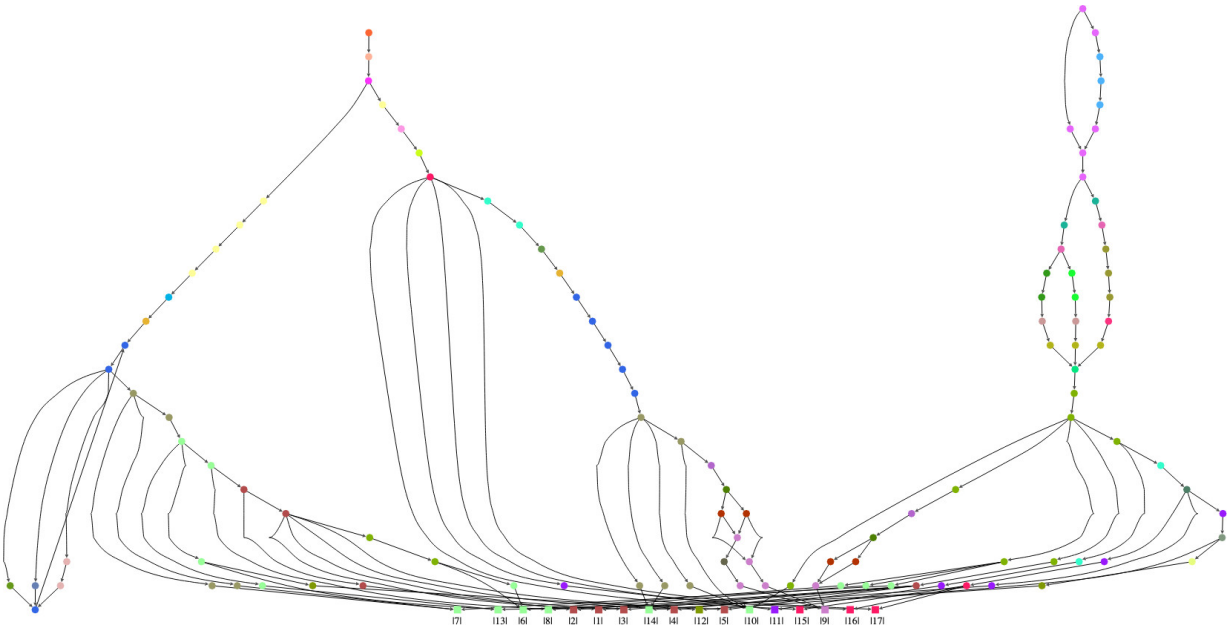

Figure A.14: Merged view of over 50 slices. Reader thread on the left, writer thread on the right. Method names are omitted. Pathfinder Explorer allows to enable and disable method names on demand to further investigate such graphs and allows to zoom in for particular slices. Numbers at the bottom correspond to reader access order of fields.

Appendix B

Instrumentation Example

```
1       private class API {
2           private RequestProcessor procedureExecutor = new RequestProcessor();
3
4           public void performPayment(Payment procedure) { // API entry point
5               TrackingScope.begin("PaymentRequest");
6               try {
7                   prepareProcedure(procedure);
8
9                   procedureExecutor.submit(procedure);
10                  procedure.await();
11
12                  finishProcedure(procedure);
13              } finally {
14                  TrackingScope.end();
15              }
16          }
17      }
18
19
20      private class RequestProcessor { // Stage 1
21          private CommitLog log = new CommitLog();
22          private Queue<Procedure> queue = new Queue<Procedure>();
23
24          public void submit(Procedure proc) {
25              proc.context = TrackingScope.getContext();
26              queue.add(proc);
27          }
28
29          private void processingLoop() {
30              while (...) {
31                  Procedure proc = queue.poll();
32                  TrackingScope.join(proc.context, "RequestProcessor");
33                  try {
34                      log.logState(proc);
35
36                      process(proc);
37
38                      proc.finish();
39                      log.logState(proc);
40                  } finally {
41                      TrackingScope.end();
42                  }
43              }
44          }
45
46      }
47
48      private class CommitLog { // Stage 2
49          private Queue<LogEntry> queue = new Queue<LogEntry>();
50
51          public void logState(Procedure proc) {
52              LogEntry w = new LogEntry(proc.getState());
53              w.context = TrackingScope.getContext();
54              queue.add(w);
55              w.await(); // wait for commit
56          }
57
```

```
58          private void processingLoop() {
59              while (...) {
60                  LogEntry entry = queue.poll();
61                  TrackingScope.join(entry.context, "CommitLog");
62                  try {
63                      writeBytes(entry.getBytes());
64                      entry.finish();
65                  } finally {
66                      TrackingScope.end();
67                  }
68              }
69          }
70      }
```

Listing B.1: An artificial multi stage request processing example instrumented with the TrackingScope API. The shown code would run within one process. A payment request traverses a request processor Stage and an asynchronous commit log Stage. The `TrackingScopeContext` is propagated end-to-end. For simplicity, TrackingScopeContext is not propagated back to callers here (e.g. from `Stage 2` back to `Stage 1`). Placing those additional TrackingScope calls would only serve to persist instrumentation decisions that are later replaced with actual request context API calls but does not influence what is tracked. With Pathfinder, 2 iterations would be needed to get to this instrumentation. In the first iteration the communication between the thread that calls `performPayment` with `Stage 1` would be revealed. In the second iteration the communication between `Stage 1` and `Stage 2` would be detected.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)