An Architecture for Universal Distributed Tracing

PhD Thesis Proposal

Jonathan Mace

October 2, 2017

Abstract of "An Architecture for Universal Distributed Tracing"

Many tracing tools have been proposed in recent research and open-source projects, to help monitor, understand, and enforce end-to-end behaviors in distributed system. Their development and deployment, however, is fraught with difficulty because they require metadata to be propagated alongside executions, including across boundaries between processes, components, and machines.

In this thesis I extend tracing tool concepts to two new areas with tools that achieve very different end-to-end goals. Retro is a resource management framework for providing end-to-end performance guarantees; it propagates tenant identifiers alongside requests, and uses them to attribute resource consumption to tenants and make per-tenant scheduling decisions. Pivot Tracing is a monitoring framework that gives users the ability to obtain metrics from one point of the system while selecting, filtering, and grouping by events from other parts of the system; it propagates partial query state alongside requests.

Next, I identify and categorize common challenges associated with developing and deploying tracing tools. This motivates the design of the Tracing Plane, an architecture for general-purpose context propagation that can be shared and reused by different tracing tools. The Tracing Plane abstracts and encapsulates context propagation components that are otherwise duplicated by most tracing tools, and decouples the design of tracing tools into separate layers that can be addressed independently by different teams of developers. To implement the Tracing Plane architecture, I propose a context representation that can support heterogeneous datatypes and is based on the theory of conflict-free replicated data types.

The potential impact of a common architecture for context propagation is significant. Developers would no longer need to make development-time decisions about which tools to deploy and support; they would no longer need a priori knowledge of dependencies between components; and it would enable more pervasive, more useful, and more diverse tracing tools.

Contents

Chapter 1

Introduction

Thesis Statement Many distributed systems monitoring and enforcement tools reason about executions end-to-end by propagating in-band metadata. We argue that this concept extends to a range of other tasks including resource management and online monitoring. Further, we propose that tracing tools be factored into layers that separate developer concerns, to make it easier to develop, deploy, and reuse tools. We demonstrate this layering by designing the Tracing Plane architecture that provides a tool-agnostic propagation layer for instrumenting system components, an intermediary data format for standardizing implementations, and a data type layer for encapsulating context behavior in the presence of concurrency.

1.1 Motivation

Many distributed systems today are shared by multiple tenants, both on private and public clouds and datacenters. These include storage, configuration management, database, queueing, and co-ordination services, such as Azure Storage [100], Amazon Dynamo [115], HDFS [218], ZooKeeper [140], and many more. Multi-tenancy has clear advantages in terms of cost and efficiency, and has in turn prompted a wide variety of architectures and frameworks, such as Netflix's microservices architecture [177], the Hadoop ecosystem [207], and similar commercial offerings [7, 129, 171].

However, monitoring, understanding, and enforcing the behavior of shared distributed systems is notoriously difficult. The potential problems are myriad: hardware and software failures, misconfiguration, hot spots, aggressive tenants, or even simply unrealistic user expectations [6, 123, 133, 151, 178]. A single request to the system — *e.g.*, an action such as loading a page on facebook.com — might entail complex executions spanning clients, networks, and distributed back-end services. Meanwhile, the symptoms of an issue can manifest in components far removed from the root cause, including in different processes, machines, and application tiers, and they may be visible to different users [133, 135].

Consequently, to diagnose problems in distributed systems one often needs to correlate and integrate data that crosses component, system, and machine boundaries, *i.e.* a user of one application may need to relate information from

some other dependent application in order to diagnose problems that span multiple systems. However, the tools and abstractions traditionally useful for diagnosing standalone programs — *e.g.* the execution stack, thread IDs, thread-local variables, debuggers, profilers, and many more — are ineffective or inadequate in this setting because they cannot coherently reason about the system when we cross boundaries between software components and machines [131, 135]. Similarly, traditional, well-studied resource isolation and management techniques in the OS and hypervisor — *e.g.* myopic thread, process, network, and IO schedulers — are insufficient for distributed systems because they lack end-to-end visibility of requests, which contend for resources both within processes and across component and machine boundaries.

Most systems today do not embed monitoring that can relate to other systems, lack end-to-end resource management, and lose important context when requests cross software component and machine boundaries. This makes it difficult to answer questions about causes of failures, uncover dependencies between components, understand performance or resource usage, and provide end-to-end performance guarantees or isolation between tenants. Many organizations face debugging, monitoring, and troubleshooting challenges after deploying distributed systems [6, 123, 133, 151, 178], and past systems have suffered cascading failures [9, 71, 133], slowdown [32, 35, 82, 133, 157], and even cluster-wide outages [9, 32, 133, 161] as a result.

In this thesis I present two *tracing tools* for distributed systems that communicate information between inter-operating system components at runtime by propagating in-band *contexts*. Next, I identify and categorize common challenges associated with developing and deploying tracing tools in distributed systems. Despite superficial differences between tracing tools, at their core they share common components and duplicate a majority of their implementation effort. Consequently, I advocate that instrumentation should be only a one-time task, reusable, and independent of any tracing tool; and that developing, deploying, and updating tracing tools should be possible without having to revisit or consider the underlying context propagation mechanisms. Finally, the principal contribution of this thesis is a layered architecture for general-purpose context propagation in distributed systems called the *Tracing Plane*. The Tracing Plane enables new tracing tools to be developed and deployed independently without having to revisit system-level instrumentation to deploy new propagation logic.

The potential impact of a common architecture for context propagation is significant. Developers would no longer need to make development-time decisions about which tools to deploy and support; they would no longer need a priori knowledge of dependencies between components; and it would enable more pervasive, more useful, and more diverse tracing tools.

1.2 Preliminary Work: End-to-End Resource Management

In this preliminary work I developed Retro, an end-to-end resource management framework for distributed systems. Retro propagates tenant identifiers alongside executions in the system and uses them to attribute resource consumption

to tenants at all points on the end-to-end execution path. Retro then co-ordinates per-tenant throttling decisions, actively throttling tenants' requests to achieve an end-to-end resource management goal, *e.g.* bottleneck fairness or guaranteed latency. Retro does not record and analyze resource consumption for offline analysis; instead it actively throttles tenants in real-time to converge to its resource management goal. Retro illustrates how a small piece of additional context propagated alongside executions – *i.e.* a tenant ID for each request – enables co-ordinated end-to-end resource management.

1.3 Preliminary Work: Dynamic Causal Monitoring

In this preliminary work I developed Pivot Tracing, a distributed systems monitoring framework. Pivot Tracing gives users, at runtime, the ability to define arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries. Pivot Tracing models the monitoring and tracing of a system as high-level queries over a dynamic dataset of distributed events, and introduces the "happened-before join" operator, $\overrightarrow{\bowtie}$, that enables queries to be contextualized by Lamport's happened-before relation, $\rightarrow$ [155]. Using $\overrightarrow{\bowtie}$, queries can group and filter events based on properties of any events that causally precede them in an execution. To track the happened-before relation between events, Pivot Tracing propagates partial query state along the execution path of each request. This enables inline evaluation of joins during request execution, drastically mitigating query overhead and avoiding the scalability issues of global evaluation.

1.4 Proposal: A Layering Abstraction for Distributed Systems Instrumentation

Retro and Pivot Tracing demonstrate how context propagation is a powerful runtime mechanism for capturing causal relationships between prior events on the execution path. In practice, however, organizations report drawn-out struggles to deploy tracing tools like Retro and Pivot Tracing, because of the high developer cost associated with *instrumentation*: the small but non-trivial modifications to system source code necessary for propagating the tool's context across request execution boundaries, *e.g.* to serialize contexts to RPC headers, to maintain them in thread-local storage, and so on. Researchers and practitioners consistently describe instrumentation as the most significant obstacle to deploying tracing tools because *all* system components must be instrumented to participate in propagating that tool's context; *e.g.* all system components must propagate Retro's tenant ID. Furthermore, most tracing tools, including Retro and Pivot Tracing, lack abstractions to enable other tools to reuse or extend existing instrumentation, so achieving end-to-end visibility is hard because it requires coherent tracing tool choices across system components at development time.

Ideally system instrumentation should be a one-time task, reusable, independent of any tracing tool, and developing, deploying, and updating tracing tools should be possible without having to revisit or consider the underlying context

propagation mechanisms. In this thesis I propose two key abstractions that, together, separate the concerns of tracing tool developers from those of system developers, and enable reusable general-purpose context propagation.

First, tracing tool developers should implement contexts using the abstraction of *execution-flow scoped variables*, which I liken to thread-local storage but dynamically scoped to end-to-end executions instead of threads. These variables are grouped into what I term *baggage*, which is a generalization of the key-value pairs used by Pivot Tracing. By reimplementing several tracing tools, I will demonstrate that this abstraction is a natural fit for tracing tool contexts, and shields tracing tool developers from the system-level context propagation required to implement them.

Second, system developers doing instrumentation should propagate opaque contexts, which I term *baggage contexts*. Baggage contexts hide their underlying context format from system developers and standardize instrumentation on a simple set of five propagation primitives, thus avoiding development-time decisions about which tracing tools to support.

To bridge these abstractions, I will seek a general-purpose intermediary context representation that must be sufficient for encoding a wide range of data types used by different tracing tools – IDs, counters, timestamps, tokens, etc. It must support these data types transparently: deploying a new tracing tool should not entail deploying propagation rules for its context. The context representation must opaquely preserve data types through all context propagation operations, two of which are particularly challenging: merging baggage instances when executions fan-in, and trimming instances when systems impose size restrictions. For many data types this is subtly complex because in the general case contexts are analogous to state-based replicated objects [212].

Finally, I propose to encapsulate both of the high-level abstractions in a layered architecture called the *Tracing Plane*. The Tracing Plane will separate the concerns of tracing tool developers from those of system developers. The intermediary layers of the tracing plane will encapsulate the intermediary context representation, and will be simple, expressive, and sufficient, I argue, to reach consensus on its deployment across systems and tracing tools; it should serve as the "narrow waist" upon which different tracing tools can be built, analogous to the role of the IP layer in networking.

Chapter 2

Related Work

2.1 Tracing Tools

Many organizations face debugging, monitoring, and troubleshooting challenges after deploying distributed systems [6, 123, 133, 151, 178]. Twitter describes *"the tried and true tools we're used to — configuration management, log processing, strace, tcpdump, etc — prove to be crude and dull instruments when applied to microservices"* [131]; for Hailo, *"rationalising the behaviour of each individual component may be simpler, but understanding the behaviour of the whole system, and ensuring correctness, is more difficult."* [135]. @Honest_Update is more candid: *"We replaced our monolith with micro services so that every outage could be more like a murder mystery."* [138].

2.1.1 Context Propagation

Inevitably, to address the aforementioned challenges distributed systems need the ability to correlate events at one point of the system with events that are meaningful from other parts of the system. This above observation has been directly or indirectly addressed in a growing body of work by maintaining the notion of a context that follows the execution patterns of applications through events, queues, thread pools, files, caches, and messages between distributed system components.

For example, node.js introduced continuation-local storage to address a growing frustration with debugging event-based javascript server code [179]. Google introduced a `context` package to Go for passing request-scoped data structures within Go servers [230]. Many systems propagate activity and trace IDs for use in debugging, profiling, and anomaly detection [126, 220]; Taint tracking and DIFC maintain and propagate security labels as the system executes, warning of or prohibiting policy violations [121, 173, 246]. Data provenance systems propagate information about the lineage of data as different components manipulate it [116, 172]. Tools for end-to-end latency, path profiling, and critical path analysis propagate partial latency measurements and information about execution paths and graphs [103, 110, 204, 228].

2.1.2 Use Cases

End-to-End Tracing Initial tracing tools in the research literature focused on end-to-end tracing, which broadly involves propagating IDs at runtime, then afterwards joining logging statements based on IDs [126, 206, 220, 232]. End-to-end tracing frameworks exist both in academia [96, 103, 126, 206, 232] and in industry [84, 120, 220, 221, 233] and have been used for a variety of purposes, including diagnosing anomalous requests whose structure or timing deviate from the norm [3, 96, 106, 107, 193, 210]; diagnosing steady-state problems that manifest across many requests [126, 206, 210, 220, 232]; identifying slow components and functions [103, 165, 220]; modelling workloads and resource usage [95, 96, 165, 232]; and explaining structural and statistical differences between 'before' and 'after' traces [165, 210]. Recent work has extended these techniques to online profiling and analysis [134, 168–170, 253].

In the open-source community, Dapper's span model is the prevailing approach with numerous derivative implementations [61, 176, 215, 222, 233, 244]. As these tracing systems have matured, their community has encountered several challenges that we describe in §2.2 [243]. This motivated the recent OpenTracing effort to standardize the semantics of this class of tracing tools [182]. Similarly, Facebook authors could not extend existing TraceID propagation to record their desired causality [109]; nor could Google authors extend Dapper [165, 193].

Other Use Cases Recent tracing tools in the research literature tackled a variety of use cases, such as measuring and predicting critical path latency [110, 141, 204, 228], tracking energy consumption [124], understanding client-server interactions [141, 159, 204], making data quality trade-offs [110], attributing resource consumption [162, 164], failure testing [136] and others [103, 146, 146]. Other work has explored similar context propagation ideas at the granularity of network packets [229].

Table A.1 highlights use cases for tracing tools deployed in production at 26 companies. Examples range from debugging individual anomalous requests, to capturing performance, resource, and latency metrics, to correlating failures and behavioral clustering. Nonetheless, there remains a lack of consensus and standards on tools for understanding, monitoring, troubleshooting, and enforcing distributed system behaviors. A recent study identified security, performance, tracing, and monitoring as highly important yet overlooked areas [6]. Despite this proliferation of tracing tools there remain significant challenges to their development and pervasive, end-to-end deployment, and distributed tracing and logging is still described as *"the most wanted and missed tool in the micro-service world"* [250].

2.1.3 Alternatives to Context Propagation

An alternative approach to propagating contexts is to infer correlation or causality from existing outputs such as logs. For example, Magpie [96] demonstrated that under certain circumstances, causality between events can be inferred after the fact. Specifically, if 'start' and 'end' events exist to demarcate a request's execution on a thread, then we can infer causality

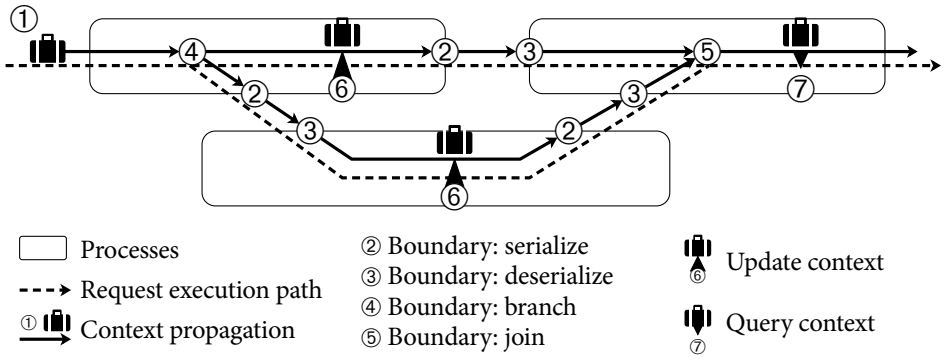


Figure 2.1: Systems propagate the tracing tool's context (①) along the execution path, including across process and machine boundaries (② ③ ④ ⑤). Tracing tools update (⑥) and query (⑦) context values.

between the intermediary events. Similarly we can also infer causality across boundaries, provided we can correlate 'send' and 'receive' events on both sides of the boundary (*e.g.*, with a unique identifier present in both events). Under these circumstances, Magpie evaluates queries that explicitly encode all causal boundaries and use temporal joins to extract the intermediary events.

In general, alternative approaches include combining identifiers – *i.e.* call ID, IP address, etc.– present across multiple logging statements [95, 109, 148, 232, 245, 251]; inferring causality using machine learning and statistical techniques [98, 165, 181, 245]; and augmenting models with static source code analysis [245, 251, 252]. These approaches avoid context propagation altogether; however, the use cases are limited to offline analysis of the information exposed by the system through logs, and they do not apply to online use cases such as scheduling and data quality trade-offs. It is also challenging to scale black-box analysis because inferring causal relationships is expensive; for example, computing a Facebook model from 1.3M traces took 2 hours for the Mystery Machine [109].

Nonetheless, these approaches have illustrated a variety of approaches to troubleshooting and analyzing distributed systems: semi-automatically honing in on root causes of performance anomalies [239]; identifying statistical anomalies [148]; online statistical outlier monitoring [93]; and analyzing critical path dependencies, slack, and speedup [194]. Manual analysis has covered an even broader range of issues, such as requests whose structure or timing deviate from the norm [3, 96, 106, 107, 193, 210]; identifying slow components and functions [103, 165, 220]; and modelling workloads and resource usage [95, 96, 165, 232].

2.1.4 Anatomy of a Tracing Tool

To motivate some of the limitations of tracing tools today, I give a brief overview of their architecture. Figure 2.1 illustrates the three main components that comprise a tracing tool. I refer to the numbers in the figure (*e.g.* ①) in my description.

Instrumentation To correlate events across different parts of the system, all system components participate by propagating a *context* (①) along the end-to-end execution path of each request (or task, job, etc.). Contexts follow the *happened-before* relation [155] between events during execution. Formally, for events a and b occurring anywhere in the system, including across system boundaries, we say that a happened before b and write $a \rightarrow b$ if the occurrence of event a causally preceded the occurrence of event b and they occurred as part of the same execution. To correctly capture the happened-before relation, contexts are copied when execution splits into concurrent branches, and merged when concurrent branches join.

Early research explored doing context propagation automatically in the operating system [104, 205]. However, the operating system's lack of visibility into application-level communication channels turned out to be a significant problem for today's distributed systems [214], which comprise a wide variety of applications, languages, and platforms. The prevailing strategy is to instead intervene at the application level, *i.e.* for system developers to directly modify source code to explicitly propagate context. This task – *instrumentation* – entails small but non-trivial interventions at all boundaries on the execution path: contexts must be serialized for inclusion with RPCs (②, ③); stored while a thread is doing work; attached to work items when queued to thread pools; duplicated when fanning out (④); merged when fanning in (⑤); etc.All distributed system components must be instrumented to propagate contexts, otherwise an intermediary component might discard context required by a later component.

Context A context comprises variables and data structures that the tracing tool uses to observe and record the happened-before relation. Examples include a single integer request ID [107,162]; IDs and flags describing recent events [126,185,220]; and data structures like sets and maps [130,164]. Contexts also need a small set of *propagation rules* to define behavior at the aforementioned execution boundaries. This includes rules for serialization and deserialization (②, ③); for duplicating contexts when executions branch (④); and for merging contexts when concurrent branches join (⑤). These rules are trivial for simple contexts such as write-once request or tenant IDs [107,162]. However for dynamic contexts these rules are more elaborate; for example, if a context comprises a set of tags then merging two contexts entails a set-union [130,164]. In the general case we formalize contexts as state-based replicated objects [212], with communication between instances occurring only when their respective execution branches join.[1]

Tracing Tool Logic Tracing tools interact with contexts intermittently during program execution, updating (⑥) and querying (⑦) values in order to reconstruct the happened-before relation between events. Most tracing tools also have background components, *e.g.* to collect logging statements or aggregate counters, but these are peripheral to our discussion.

Like instrumentation, developers must intervene in the system's source code to deploy tracing tool logic. However,

[1] Out-of-band communication would violate the happened-before relation.

unlike instrumentation, the extent of developer intervention depends on the tool at hand; for example, distributed logging scatters logging statements throughout all components, whereas resource accounting adds counters to only very specific resource APIs. Some tools, such as security auditing, only deploy tracing tool logic to a subset of components; *e.g.* to set the context's user ID in the front end, and to check the user ID once the request reaches the database [6, 220, 249]. It is important to distinguish tracing tool modifications from those of instrumentation, because instrumentation is always pervasive — for a context to propagate between any two points in the system, all intermediary components must be instrumented.

2.2 Tracing Tool Deployment

Today there remains a lack of consensus and standards on tracing tools for distributed systems. Despite a proliferation of tracing tools there remain significant challenges to their development and pervasive, end-to-end deployment.

2.2.1 Cognitive Load

Although §2.1.4 describes tracing tools as comprising three components, in practice there is little to no separation between the three. To perform any one task – system instrumentation; deploying tracing tool logic; or even designing the tracing tool – is cognitively challenging, because it requires tacit understanding of all components: the system being instrumented; the semantics of the tracing tool; and context subtleties when executions branch and join.

Concretely, to propagate and interact with contexts, developers must pay attention to serialization, encoding schemes, and binary formats. They must also know the libraries (*e.g.* RPCs) and concurrency structures used by the system, such as event loops, futures, and queues, to determine boundaries for propagation. The semantics of the tracing tool greatly affect instrumentation, because they are usually deployed together; not only does this entail understanding tracing tool logic at instrumentation time, but also, by specializing instrumentation, precludes reuse of the instrumentation even by similar tools. For example, HDFS, HBase, and Phoenix, instrumentation is hard-coded to HTrace contexts and rules [61, 73, 91]. Zipkin omits merge rules [183, 184], leading to difficulties instrumenting queues [243], asynchronous executions [187, 188, 190] and capturing multiple-parent causality [189, 191]. HDFS and HBase developers encountered similar problems due to HTrace lacking rules to capture multiple-parent causality [54, 69, 77, 81]. When tracing tools do not require instrumentation of all boundaries, it sows confusion among developers about whether to propagate contexts across those boundaries; the most common example being RPC response instrumentation [83, 186, 225]. Developers also struggle to instrument execution patterns when they do not fit into the tracing tool's intended model. For example, Dapper's request-response span model is ill-suited for instrumenting streams, queues [243], async [187, 188, 190], and several others [69, 77, 189, 191].

Because there are no pre-existing abstractions or implementations for context propagation, it is insufficient to

simply state "propagate this data structure" in a setting where executions arbitrarily branch and join. Instead, something seemingly simple like an integer can entail complex propagation rules in order to behave correctly and consistently; *e.g.* if it is used as a counter, its underlying implementation will more closely resemble a version vector [199]. An example of this is Pivot Tracing [164]; to propagate sets of tuples requires a complicated versioning scheme based on interval tree clocks [5].

2.2.2 Instrumentation Burden

Researchers and practitioners consistently describe instrumentation as the most time consuming and difficult part of deploying a tracing tool [109,125,126,147,209,219]. from early works like X-Trace: *"capturing causal connections presented the highest variability in difficulty"* [125,126]; to recent efforts such as OpenTracing: *"deploying a distributed tracing system requires person-years of engineer effort, monkey-patched communication packages, and countless inconsistencies across platforms"* [219]. Instrumentation is a challenge because system modifications are dispersed across a wide range of disparate source code locations, making it brittle and easy to get wrong [220]. For example, developers might inadvertently omit infrequently exercised code paths such as background tasks, edge-case handling, and failure recovery; or only instrument specific request types. In Cassandra [21], missing instrumentation for request retries caused traces to end prematurely after the first attempt [113], and missing instrumentation for paginated requests caused traces to end after the first page of results [112]; initial efforts in HDFS [73] only instrumented I/O operations, so developers had to later revisit instrumentation for metadata operations [78] and background tasks [68]. In many cases, missing or misaligned instrumentation will prematurely discard request contexts [27,28,51,55,85,224]; on the other hand, failing to discard contexts enables them to linger and associate with the next request, causing *e.g.* distinct traces to merge [14] or misattributed resource consumption [162,163]. Aligning instrumentation between a system and its dependencies is also challenging; for example, Accumulo developers blamed HTrace's wrapper library for an instrumentation bug [86], discovered it was actually their own fault [13], and introduced new bugs attempting to fix it [17,18].

Persistent problems may arise intermittently after an instrumented system is deployed. Unrelated code changes can break instrumentation if they fail to propagate contexts across new or modified execution boundaries; for example, a patch to Hadoop's inter-process communication [33] omitted HTrace context in the new call headers [34]; refactored components in Cassandra [22] failed to propagate request contexts across new execution boundaries [24,25]; and in HBase, a patch to the write-ahead log led to gaps in traces due to instrumentation omissions [40]. When a system is only partially instrumented, the overlap between instrumented and uninstrumented code paths makes revisiting the instrumentation difficult. For example, Cassandra developers extended instrumentation to background tasks and found that *"It was more involved than I thought, partly because of heisenbugs and the trace state mysteriously not propagating"* [26].

Testing instrumentation for end-to-end correctness is a further challenge because it touches so many components;

the result is poor test coverage, complicated integration tests, broken tests across application versions, and a struggle to ensure that instrumentation remains correct [25, 43, 52, 53, 80].

Some systems mitigate the instrumentation burden by propagating context in shared underlying communication and concurrency layers instead of directly at the application level [220]. Fonseca *et al.* proposed alleviating X-Trace instrumentation difficulties by modifying concurrency libraries or runtime environments [126]. Dapper [220] stated this as a design goal and mostly avoided application-level instrumentation by instead modifying Google's core threading, control flow, and RPC libraries. Recent work has also used source and binary rewriting techniques to automatically instrument common execution patterns [141, 162–164].

However, this is not always feasible; extreme heterogeneity is commonplace, as many large organizations have to deal with legacy components and decentralized, disparate development teams [209]. For example, Facebook systems comprise a broad range of languages, middleware, and execution environments [109]. Instrumentation effort is proportional to system heterogeneity: the wider the variety of languages and platforms in use, the more effort required to do instrumentation [160, 209]; for systems like Facebook's, adding instrumentation is a *"Herculean"* task [109]. The result of these complications is that instrumentation is done only once, if at all, and ends up being tied to a particular tracing tool.

2.2.3 End-to-End Deployment

Deploying tracing tools end-to-end is challenging because it requires coherent choices and participation across all system components. However, developers of different systems and components are often isolated from one another, causing them to make incompatible or conflicting choices about which tracing tools deploy; *e.g.* at Pinterest, *"enabling and deploying instrumentation across several services is like herding cats"* [147]. Amazon required extensive, co-ordinated effort to deploy X-Ray [97] across dependent systems to support tracing in Lambda [1, 8].

In practice, many systems lack coherent end-to-end tools; tracing tools will lose visibility of executions once they enter a system that did not deploy the same tool, leading to conflicts because of incompatible tracing systems. For example, the Hadoop ecosystem [207] has three distributed databases based on BigTable's design [105] and each database implemented its own an ad-hoc tracing system: Accumulo [11] developed CloudTrace [20]; HBase [39] developed HTrace [60, 61]; and Cassandra [21] developed QueryTracing [23]. Accumulo developers wanted CloudTrace to extend to the underlying distributed file system HDFS [12]; however, HDFS developers opted for compatibility with HBase and deployed HTrace instead [73]. As a result, to get visibility of HDFS, Accumulo developers replaced CloudTrace with HTrace [19].

However, migrating between tracing tools is easier said than done. Instrumentation is often hard-coded to the variables and propagation logic of the tracing tool; *e.g.*, binding to request [107], workflow [162], or span [87, 233] IDs by name. Consequently, changing tools entails revisiting the instrumentation to update variables, propagation logic, and tracing tool API invocations. In the prior example, migrating Accumulo to HTrace meant updating instrumentation at

471 source code locations [19]. Similarly extensive changes were required to deploy Zipkin in Cassandra [22, 30, 241] and Phoenix [92].

Different versions of the same tracing tool suffer similar problems if they change their context format or propagation logic [16, 26, 50, 223, 226]. Developers must be careful to consider compatibility implications when updating tools, especially if they wish to support tracing during upgrades, incremental upgrades, or mixed versions. For example, hard-coded serialization logic in Cassandra could cause out-of-bounds errors during deserialization if a newer version of the application extends the context definition [26]; similarly, clients of the tracing system Sleuth would crash when encountering unexpected context values [223, 226]. Accumulo developers were hesitant to change HTrace versions due to possible Hadoop conflicts [15, 16, 50], expressing frustration with the *"continued lack of concern in the HTrace project around tracing during upgrades"* [16]. The effort needed to update a tool's version is often the same as simply migrating to a different tool; for example, systems updating HTrace versions each required hundreds of changes in dozens of files [15, 65, 81]. Accumulo developers decried the *"continued lack of concern in the HTrace project around tracing during upgrades"* [16].

Some tools alleviate these issues with ad-hoc compatibility shims, so that a system instrumented for a different tool can share contexts and talk to the same backends. This approach is fragile even for tracing tools that ostensibly perform the same task. For example, open-source Dapper derivatives preserve causal relationships at different granularities, leading to "severe signal loss" when integrating with less expressive tools [184], or requiring system-level changes to capture missing relationships required by more expressive tools [15, 54, 81].

2.3 Monitoring Distributed Systems

Most monitoring and diagnosis tools commonly used today – logs, counters, and metrics – face several fundamental limitations, which I outline in more detail in this section.

2.3.1 Existing Approaches to Monitoring

One Size Does Not Fit All By default, the information required to diagnose an issue may not be reported by the system or contained in system logs. Current approaches tie logging and statistics mechanisms into the development path of products, where there is a mismatch between the expectations and incentives of the developer and the needs of operators and users. Panelists at SLAML [99] discussed the important need to *"close the loop of operations back to developers"*. According to Yuan *et al.* [248], regarding diagnosing failures, *"(…) existing log messages contain too little information. Despite their widespread use in failure diagnosis, it is still rare that log messages are systematically designed to support this function."*

This mismatch can be observed in the many issues raised by users on Apache's issue trackers: to request new

metrics [37, 41, 46–48, 64, 74]; to request changes to aggregation methods [49, 63, 66]; and to request new breakdowns of existing metrics [36, 44, 45, 56–59, 62, 63, 70, 72, 76, 79, 89]. Many issues remain unresolved due to developer pushback [57, 62, 72, 74, 79] or inertia [44, 46, 47, 64, 66, 70, 76, 89]. Even simple cases of misconfiguration are frequently unreported by error logs [247].

Costs of Monitoring Eventually, applications may be updated to record more information, but this has effects both in performance and information overload. Users must pay the performance overheads of any systems that are enabled by default, regardless of their utility. For example, HBase SchemaMetrics were introduced to aid developers, but all users of HBase pay the 10% performance overhead they incur [63]. The HBase user guide [67] carries the following warning for users wishing to integrate with Ganglia [166]: *"By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase."*

The glut of recorded information presents a "needle-in-a-haystack" problem to users [202]; while a system may expose information relevant to a problem, *e.g.* in a log, extracting this information requires system familiarity developed over a long period of time. For example, Mesos cluster state is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state [90].

Beyond Metrics and Logs A variety of tools have been proposed in the research literature to complement or extend application logs and performance counters. These include the use of machine learning [148, 174, 181, 245] and static analysis [252] to extract better information from logs; automatic enrichment of existing log statements to ease troubleshooting [248]; end-to-end tracing systems to capture the happened-before relationship between events [126, 220]; state-monitoring systems to track system-level resources and indicate the health of a cluster [166]; and aggregation systems to collect and summarize application-level monitoring data [153, 234]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [240]. These tools suffer from the aforementioned challenges of pre-defined information.

Dynamic Instrumentation Dynamic instrumentation frameworks such as Fay [122], DTrace [102], and SystemTap [200] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime, and have proven extremely useful in the diagnosis of complex and subtle system problems [101]. Because of their side-effect-free nature, however, they are limited in the extent to which probes may share information with each other. In Fay, only probes in the same address space can share information, while in DTrace the scope is limited to a single operating system instance.

2.3.2 Cross-Component Monitoring

Crossing Boundaries In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [58] outlines how MapReduce lacks the ability to access HBase metrics on a per-task basis, and that the framework only returns aggregates across all tasks. MESOS-1949 [89] outlines how the executors for a task do not propagate failure information, so diagnosis can be difficult if an executor fails. In discussion the developers note: *"The actually interesting / useful information is hidden in one of four or five different places, potentially spread across as many different machines. This leads to unpleasant and repetitive searching through logs looking for a clue to what went wrong. (…) There's a lot of information that is hidden in log files and is very hard to correlate."*

End-to-End Tracing Prior research has presented mechanisms to observe or infer the relationship between events and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [180, 202]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [220], Apache's HTrace [84], Accumulo's Cloudtrace [11], and Twitter's Zipkin [233]. As described in §2.1.2, these approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems outlined in §2.3.1 concerning what to record.

VScope [239] introduces a novel mechanism for honing in on root causes on a running system, but at the last hop defers to offline user analysis of debug-level logs, requiring the user to trawl through 500MB of logs which incur a 99.1% performance overhead to generate. While causal tracing enables coherent sampling [208, 220] which controls overheads, sampling risks missing important information about rare but interesting events.

2.4 Resource Management in Distributed Systems

Many shared distributed systems implement some variant of performance isolation, such as fair sharing [127, 217, 231], throttling aggressive tenants [100, 218], and providing latency or throughput guarantees [115, 231, 237, 238].

2.4.1 Need for Fine-Grained Resource Isolation

When tenants compete inside a process, traditional and well-studied resource management techniques in the operating system and hypervisor are unsuitable for protecting tenants from each other. In such cases, aggressive tenants can overload the process and gain an unfair share of resources. In the extreme, this lack of isolation can lead to a denial-of-service

to well-behaved tenants and even system wide outages. For example, eBay Hadoop clusters regularly suffered denial of service attacks caused by heavy users overloading the shared HDFS NameNode [157,158]. HDFS users report slowdown for a variety of reasons: poorly written jobs making many API calls [82]; unmanaged, aggressive background tasks making too many concurrent requests [71]; and computationally expensive APIs [35]. Impala [154] queries can fail on overloaded Kudu [161] clusters due to request timeouts and a lack of fair sharing [88]. Cloudstack users can hammer the shared management server, causing performance issues for other users or even crashes [32]. Guo et al. [133] describe examples where a lack of resource management causes failures that cascade into system-wide outages: a failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading servers; overloaded servers not responding to heartbeats, triggering further data replication and overload.

2.4.2 Multi-Resource Scheduling

Several research projects tackle multi-resource allocation, such as Cake [237], IOFlow [231], and SQLVM [175]. Cake provides isolation between low-latency and high-throughput tenants using HBase and HDFS. However, it treats HDFS as a single resource, and cannot target specific resource bottlenecks and workflows that overload these resources. IOFlow provides per-tenant guarantees for remote disk IO requests in datacenters but does not schedule other resources such as threadpools, CPU, and locks. SQLVM [175] provides isolation for CPU, disk IO, and memory for multiple relational databases deployed in a single machine, but does not deal with distributed scenarios.

In the data analytics domain, task schedulers such as Mesos [137], Yarn [236], or Sparrow [195] use an admission control approach to allocate individual tasks to machines. In these frameworks, each task passes through the scheduler before starting its execution, the scheduler can place it to an arbitrary machine in the cluster and after starting execution, the task is not scheduled any more. In typical distributed systems, requests do not pass through a single point of execution and routing of a request through the system is driven by complex internal logic. Finally, to achieve fine-grained control over resource consumption, requests have to be throttled *during* its execution, not only at the beginning. These frameworks thus do not directly apply to scheduling in general distributed systems.

2.4.3 Ad-Hoc Approaches to Resource Isolation

In all cases observed, the enforcement mechanisms for high-level policies were manually implemented. For example, Cake [237] manually instruments the RPC entry points of HDFS and HBase to add queues and associates tenants based on an identifier from the HDFS RPC headers; IOFlow [231] modifies queues in key resources (*e.g.*, NIC, disk driver) on the data path; and Pisces [217] modifies the scheduling and queueing code of Membase and directly updates tenant weights at these queues.

Lack of visibility of actual resource bottlenecks leads to the ad-hoc selection of the metrics used for performance

isolation. For example, Azure Storage [100] and Pisces [217] select only request rate and operation size as metrics. SQLVM [175] uses CPU, I/O, and memory as key resources. Cake [237] breaks HDFS requests into equal-sized chunks, then assumes disk as a bottleneck and uniform cost for each chunk.

Given the burden on application programmers, inevitably, many distributed systems do not provide isolation between tenants, or only utilize ad-hoc isolation mechanisms to address individual problems reported by users. For example, HDFS recently introduced priority queueing [38] to address the problem that *"any poorly written MapReduce job is a potential distributed denial-of-service attack,"* but this only provides coarse-grained throttling of aggressive users over long periods of time. CloudStack addressed denial-of-service attacks in release 4.1, adding manually configurable upper bounds for tenant request rates [31]. A recent HBase update [42] introduced rate limiting for operators to throttle aggressive users, but it relies on hard-coded thresholds, manual partitioning of request types, and lacks cost-based scheduling. In these examples, the developers identify multi-tenant fairness and isolation as an important, but difficult, and as-yet unsolved problem [29, 88, 211].

2.4.4 End-to-end (resource) tracing

Banga and Druschel addressed the mismatch between OS abstractions and the needs of resource accounting with resource containers [94], which, albeit in a single machine, aggregate resource usage orthogonally to processes, threads, or users. Causeway [104] modifies Linux to propagate generic metadata when threads communicate, and uses this to build meta-applications that could include resource accounting. Whodunit [103] uses causal propagation to record timings between parts of a program, and provides a profile of where requests spent their time. Timecard [204] also propagates cumulative time information in the request path between a mobile web client and a server, and uses this in real time to speed up the processing of requests that are late.

Chapter 3

Preliminary Work: End-to-End Resource Management

In this chapter I introduce Retro, a resource management framework for multi-tenant distributed systems. Retro monitors per-tenant resource usage both within and across distributed systems, and exposes this information to centralized resource management policies through a high-level API. A policy can shape the resources consumed by a tenant using Retro's control points, which enforce sharing and ratelimiting decisions. I demonstrate Retro through three policies providing bottleneck resource fairness, dominant resource fairness, and latency guarantees to high-priority tenants, and evaluate the system across five distributed systems: HBase, Yarn, MapReduce, HDFS, and Zookeeper.

3.1 Hadoop Architecture

Before describing the challenges of resource management in multi-tenant distributed systems, I first give a high-level overview of Hadoop components. Though this work is presented in the context of the Hadoop stack, the results I present generalize to other distributed systems as well.

Figure 3.1 shows the relevant components of the Hadoop stack. HDFS [218], the distributed file system, consists of DataNodes (DN) that store replicated file blocks and run on each worker machine, and a NameNode (NN) that manages the filesystem metadata. Yarn [236] comprises a single ResourceManager (RM), which communicates with NodeManager (NM) processes on each worker. Hadoop MapReduce is an application of Yarn that runs its processes (application master and map and reduce tasks) inside Yarn containers managed by NodeManagers. HBase [39] is a data store running on top of HDFS that consists of RegionServers (RS) on all workers and an HBase Master, potentially co-located with the NameNode or Yarn. Finally, ZooKeeper [140] is a system for distributed coordination used by HBase.

MapReduce job input and output files are loaded from HDFS or HBase, but during the job's *shuffle* phase, intermediate output is written to local disk by mappers (bypassing HDFS) and then read and transferred by NodeManagers to reducers. Reading and writing to HDFS has the NameNode on the critical path to obtain block metadata. An HBase query executes on a particular RegionServer and reads/writes its data from one or many DataNodes.

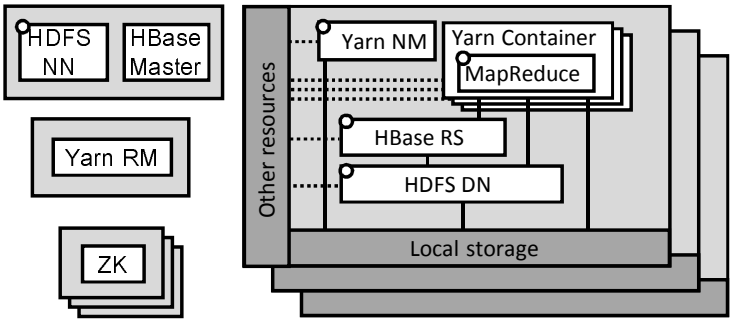


Figure 3.1: Typical deployment of HDFS, ZooKeeper, Yarn, MapReduce, and HBase in a cluster. Gray rectangles represent servers, white rectangles are processes, and white circles represent control points that we added. See text for details.

In such deployment, a large number of processes share the hardware resources on worker nodes. Moreover, each process might concurrently execute requests on behalf of multiple users or background tasks.

3.2 Resource Management Challenges

In §1.1 I introduced the overarching challenges that make it difficult to monitor, troubleshoot, and enforce distributed systems. In this section, I elaborate on four additional challenges that make resource management specifically challenging.

Any resource can become a bottleneck Figure 3.2 demonstrates how the latency of an HDFS client can be adversely affected by other clients executing very different types of requests, contending for different resources. In production, a Hadoop job that reads many small files can stress the storage system with disk seeks, as workload A in the figure, and impact all other workloads using the disks. Similarly, a workload that repeatedly resubmits a job that fails quickly puts a large load on the NameNode, like workload C, as it has to list all the files in the job input directories. In communication with Cloudera [10], they acknowledge several instances of aggressive tenants impacting the whole cluster, saying "anything you can imagine has probably been done by a user". Interviews with service operators at Microsoft confirm this observation.

Multiple granularities of resource sharing On the one hand, concurrently executing workflows share software resources, such as threadpools and locks, within a process, while on the other hand, resources, such as the disk on Hadoop worker nodes, are distributed across the system. The disk resource, for example, is accessed by DataNode, NodeManager, and mapper/reducer processes running across all workers. Systems have many entry points (*e.g.*, HBase, HDFS, or MapReduce API) and maintenance tasks are launched from inside the system. Finally, enforcing resource usage for long-running requests requires throttling inside the system, not just at the entry points.

Maintenance and failure recovery cause congestion Many distributed systems perform background tasks that are not directly triggered by tenant requests but compete for the same resources. For example, HDFS performs data replication

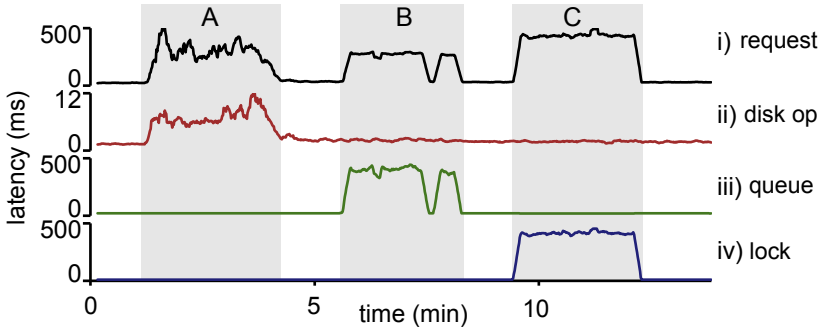


Figure 3.2: i) Average request latency for a client reading 8kB files from HDFS [218] is impacted by different workloads that A: replicate HDFS blocks; B: list large directories; and C: make new directories; these overload the disk, threadpool, and locks respectively. ii) latency of DataNode disk operations, iii) latency at NameNode RPC queue, iv) latency to acquire NameNode "NameSystem" lock.

after failures, asynchronous garbage collection after file deletion, and block movement for balancing DataNode load. In some cases, these background tasks can adversely affect the performance of foreground tasks. For example, HDFS-4183 [71] describes an example where a large number of files are abandoned without closing, triggering a storm of block recovery operations after the lease expiration interval one hour later, which overloads the NameNode. Guo et al. [133] describe a similar failure in Microsoft's datacenter where a background task spawned a large number of threads, overloading the servers. On the other hand, some of these tasks need to be protected *from* foreground tasks. For example, Guo et al. [133] describe a cascading failure resulting from overloaded servers not responding to heartbeats, triggering further data replication and further overload.

Resource management is nonexistent or noncomprehensive Systems like HDFS, ZooKeeper, and HBase do not contain any admission control policies. While Yarn allocates compute slots using a fair scheduler, it ignores network and disk, thus, an aggressive job can overload these resources. Interviews with service operators at Microsoft indicate that productions system often implement resource management policies that ignore important resources and use hardcoded thresholds. For example, a policy might assume that an `open()` is 2x more expensive than `delete()`, while the actual usage varies widely based on parameters and system state, resulting in very inaccurate resource accounting. The policies are often tweaked manually, typically *after* causing performance issues or outages, or when the system or the workloads change. Writing the policies often requires intimate knowledge of the system and of the request resource profile, which may be impossible to know a priori.

3.3 Design

The main goal of Retro is to enable simple, targeted, system-agnostic, and resource-agnostic resource-management polices for multi-tenant distributed systems. Examples of such policies are: a) throttle aggressive tenants who are getting

an unfair share of bottlenecked resources, b) shape workflows to provide end-to-end latency or throughput guarantees, or c) adjust resource allocation to either speed up or slow down certain maintenance or failure recovery tasks.

Retro addresses the resource management challenges described in §3.2 by separating the mechanisms of measurement and enforcement of resource usage from high-level, global resource management policies. It does this by using three unifying abstractions – *workflows*, *resources*, and *control points* – that enable logically centralized policies to be succinctly expressed and apply to a broad class of resources and systems.

3.3.1 Retro abstractions

Workflow Resource contention in a distributed system can be caused by a wide range of system activities. Retro treats each such activity as a first-class entity called a *workflow*. A workflow is a set of requests that forms the unit of resource measurement, attribution, and enforcement in Retro. For instance, a workflow might represent requests from the same user, various background activities (such as heartbeats, garbage collection, or data load balancing operations), or failure recovery operations (such as data replication). The aggregation of requests into a workflow is up to the system designer. For instance, one system might treat all background activities as one workflow but another might treat heartbeats as a distinct workflow from other activities, if the system designer decides to provide a different priority to heartbeats.

Each workflow has a unique workflow ID. To properly attribute resource usage to individual workflows, Retro propagates the workflow ID along the execution path of all requests. This causal propagation [126, 206, 220, 232] allows Retro to attribute the usage of a resource to a workflow at any point in the execution, whether within a shared process or across the network.

Resources A comprehensive resource management policy should be able to respond to a contention in any resource – hardware or software – and attribute load to workflows using it. A key hypothesis of Retro is that resource management policies can and should treat all resources, from thread pools to locks to disk, uniformly under a common abstraction. Such a uniform-treatment allows one to state policies that respond to disk contention, say, in the same way as lock contention. Equally importantly, this allows gradually expanding the scope of resource-management to new resources without policy change. For instance, a storage service might start by throttling clients based on their network or disk usage. However, as the complexity of the service increases to include sophisticated meta-data operations, the service can start throttling by CPU usage or lock-contention. On the other hand, the challenge in providing such a unifying abstraction is to capture the behavior of varied kinds of resources with different complex non-linear performance characteristics.

To overcome this challenge, Retro captures a resource's current *first-order* performance with two unitless metrics:

- **slowdown** indicates how slow the resource is currently, compared to its baseline performance with no contention;
- **load** is a per-workflow metric that determines who is responsible for the slowdown.

As a simple example, consider an abstract resource with an (unbounded) queue. Let $Q_{w,i}$ be the queueing time of

the ith request from workflow w in a time interval and let $S_{w,i}$ be the time the resource takes to service that request. During this interval, the load by w is $\Sigma_i S_{w,i}$ and the slowdown is $\Sigma_{w,i}(Q_{w,i} + S_{w,i})/\Sigma_{w,i} S_{w,i}$. Note, the denominator of the slowdown is the time taken to process the requests if the queue is empty throughout the interval.

The reactive policies in Retro allow these metrics to provide a linear approximation of the complex non-linear behavior. The policies continuously measure the resource metrics while making incremental resource allocation changes. Operating in such a feedback loop enables simple abstractions while reacting to nonlinearities in the underlying performance characteristics of the resource.

Resources in real systems are more complex than the simple queue above. Retro's goal is to hide the complexities of measuring the load and slowdown of different resources in *resource libraries* that are implemented once and reused across systems. See §3.4.2 for details.

An important implication of this abstraction is that it is not possible to query the capacity of a resource. Instead, a policy can treat a resource to have reached its capacity if the slowdown exceeds some fixed constant. Directly measuring true capacity is often not possible because of many request types supported (*e.g.* open, read, sync, etc. on a disk) and because of effects of caching or buffering, workflow demands do not compose linearly. Also, due to limping hardware [118], estimating the current operating capacity is next to impossible.

Control points To separate the low-level complexities of enforcing resource allocation throughout the distributed system, we introduce the *control point* abstraction. A control point is a point in the execution of a request where Retro can enforce the decisions of resource scheduling policies. Each control point executes locally, such as delaying requests of a workflow using a token bucket, but is configured centrally from a policy.

While a control point can be placed directly in front of a resource (such as a thread pool queue), it can more generally be located anywhere it is reasonable to sleep threads or delay requests, such as in HDFS threads sending and receiving data blocks. The location of control points should be selected by the system designer while keeping a few rules in mind. A control point should not be inserted where delaying a request can directly impact other workflows, such as when holding an exclusive lock. Conversely, some asynchronous design patterns (such as thread pools) present an opportunity to interpose control points, as it is unlikely that a request will hold critical resources yet potentially block for a long period of time.

Each logical control point has one or more instances. A point with a single instance is centralized, such as a point in front of the RPC queue in HDFS NameNode. Distributed points, such as in the DataNode or its clients, have many, potentially thousands of instances. Each instance measures the current, per-workflow throughput which is aggregated inside the controller.

To achieve fine-grained control, a request has to periodically pass through control points, otherwise, it could consume unbounded amount of resources. To illustrate this, consider a request in HBase that scans a large region, reading data

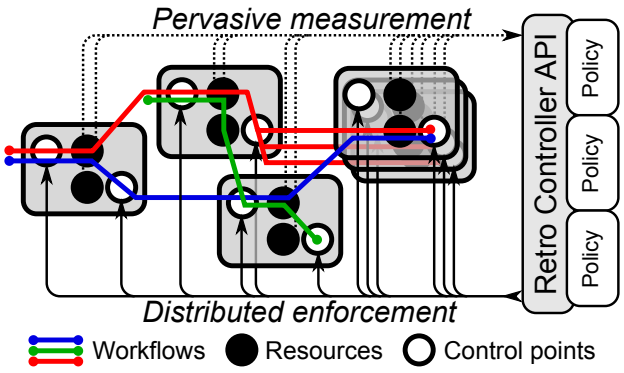


Figure 3.3: Retro architecture. Gray boxes are system components on the same or different machines. Workflows start at several points and reach multiple components. Intercepted resources (●) generate measurements that serve as inputs to policies. Policy decisions are enforced by control points (○).

from multiple store files in HDFS. If Retro only throttles the request at the RegionServer RPC queue, a policy has only one chance to stop the request; once it enters HBase, it can read an unbounded amount of data from HDFS and perform computationally expensive filters on the data server-side. By adding a point to the DataNode block sender, we can control the workflow at the granularity of 64kB HDFS data packets. More generally, the longer the period of time a request can execute without passing through a control point, the longer it will take any policy to react. This is similar to the dependence between the longest packet length L_{max} and the fairness guarantees provided by packet schedulers [197, 227].

3.3.2 Architecture

Figure 3.3 outlines the high-level architecture of Retro and its three main components. First, Retro has a measurement infrastructure that provides near-real-time resource usage information across all system resources and components, segmented by workload. Second, the logically centralized controller uses the resource library to translate raw measurements to the load and slowdown metrics, and provides them as input to Retro policies. Third, Retro has a distributed, coordinated enforcement mechanism that consistently applies the decisions of the policies to control points in the system. We discuss the design of the controller in the following paragraphs. In §3.4 we describe the measurement and enforcement mechanisms in detail, and in §3.5 we present the implementation of three policies.

Logically centralized policies In current systems, resource management policies are hard-coded into the system implementation making it difficult to maintain as the system and policies evolve. A key design principle behind Retro is to separate the mechanisms (§3.4) from the policies (§3.5). Apart from making such policies easier to maintain, such a separation allows policies to be reused across different systems or extended with more resources.

Borrowing from the design of Software Defined Networks and IOFlow [231], Retro takes the separation a step further by logically centralizing its policies. This makes policies much easier to write and understand, as one does not have to worry about myopic local policies making conflicting decisions. In this light, we can view Retro as building a "control

`workflows()`	list of workflows
`resources()`	list of resources
`points()`	list of throttling points
`load(r,w)`	load on `r` by workflow `w`
`slowdown(r)`	slowdown of resource `r`
`latency(r,w)`	total latency spent by `w` on `r`
`throughput(p,w)`	throughput of workflow `w` at point `p`
`get_rate(p,w)`	get the throttling rate of workflow `w` at point `p`
`set_rate(p,w,v)`	throttle workflow `w` at point `p` to `v`

Table 3.1: Retro API used by the scheduling policies. We omit auxiliary calls to set, for example, the reporting interval and smoothing parameters, as well as to obtain more details such as operation counts, etc.

plane" for distributed systems, and providing a separation of concerns for policy writers and system developers and instrumenters.

Retro exposes to policies a simple API, shown in Table 3.1, that abstracts the complexity of individual resources and allows one to specify resource-agnostic scheduling policies, as demonstrated in §3.5. The first three functions in the table correspond to the three abstractions explained above. In addition, `latency(r,w)` returns the total time workflow `w` spent using resource `r`. `throughput(p,w)` measures the aggregate request rate of workflow `w` through a (potentially distributed) throttling point `p`, such as the entry point to the RS process. Finally, policies can affect the system through Retro's throttling mechanisms.

3.4 Implementation

3.4.1 Per-workflow resource measurement

End-to-end ID propagation At the beginning of a request, Retro associates threads executing the request with the workflow by storing its ID in a thread local variable; when execution completes, Retro removes this association. While the developer has to manually propagate the workflow ID across RPCs or in batch operations, we use AspectJ to automatically propagate the workflow ID when using `Runnable`, `Callable`, `Thread`, or a `Queue`.

Aggregation and reporting When a resource is intercepted, Retro determines the workflow associated with the current thread, and increments in-memory counters that track the per-workflow resource use. These counters include the number of resource operations started and ended, total latency spent executing in the resource and any operation-specific statistics such as bytes read or queue time. When the workflow ID is not available, such as when parsing an RPC message from the network, the resource use is attributed to the *next* ID that is set on the current thread (*e.g.*, after extracting the workflow ID from the RPC message). Retro does not log or transmit individual trace events like X-Trace or Dapper, but only aggregates counters in memory. A separate thread reads and reports the values of the counters to the central controller at

a fixed interval, currently once per second. Reports are serialized using protocol buffers [235] and sent using ZeroMQ [4] pub-sub. The centralized controller aggregates reports by workflow ID and resource, smoothes out the values using exponential running average, and uses the resource library to compute resource load and slowdown.

Batching In some circumstances, a system might batch the requests of multiple workflows into a single request. HDFS NameNode, HBase RegionServers, and ZooKeeper each have a shared transaction log on the critical path of write requests. In these cases, we create a *batch workflow ID* to aggregate resource consumption of the batch task (e.g., the resources consumed when writing HBase transaction logs to HDFS). Constituent workflows report their relative contributions to the batch (e.g., serialized size of transaction) and the controller decomposes the resources consumed by the batch to the contributing workflows.

Automatic resource instrumentation using AspectJ Retro uses AspectJ [149] to automatically instrument all hardware resources and resources exposed through the Java standard library. Disk and network consumption is captured by intercepting constructor and method calls on file and network streams. CPU consumption is tracked during the time a thread is associated with a workflow. Locking is instrumented for all Java monitor locks and all implementers of the `Lock` interface, while thread pools are instrumented using Java's `Executors` framework. The only manual instrumentation required is for *application-level* resources created by the developer, such as custom queues, thread pools, or pipeline processing stages.

AspectJ is highly optimized and *weaves* the instrumentation with the source code when necessary without additional overheads. In order to avoid potentially expensive runtime checks to resolve virtual function calls, Retro instrumentation only intercepts constructors to return proxy objects that have instrumentation in place.

3.4.2 Resource library

Retro presents a unified framework that incorporates individual models for each type of resource. Management policies only make incremental changes to request rates allocated to individual workflows; for example, if the CPU is overloaded, a policy might reduce total load on the CPU by 5%. Therefore, as long as we correctly detect contention on a resource, iteratively reducing load on that resource will reduce the contention. Our models, thus, capture only the first-order impact of load on resource slowdown.

CPU We query the per-thread CPU cycle counter when setting and unsetting the workflow ID on a thread (using `QueryThreadCycleTime` in Windows and `clock_gettime` in Linux) to count the total number of CPU cycles spent by each workflow. The load of a workflow is thus proportional to its usage of CPU cycles. To estimate the slowdown, we divide the actual latency spent using CPU by the *optimal latency* of executing this many cycles at the CPU frequency. That is, on

a core with a frequency of F cycles per second, a workflow that consumes f cycles in t seconds has slowdown:

$$\text{slowdown} = \frac{F \times t}{f} \tag{3.1}$$

Since part of the thread execution could be spent in synchronous IO operations, we only use CPU cycles and latency spent outside of these calls to compute CPU slowdown. If frequency scaling is enabled, we could use other existing performance counters to detect CPU contention [242].

Disk To estimate disk slowdown, we use a subset of disk IO operation types that we monitor, in particular, `reads` and `syncs`. For example, given a time interval with n `syncs` and b bytes written during these operations, we use a simple disk model that assumes a single seek with duration T_s for each `sync`, followed by data transfer at full disk bandwidth B. Denoting t to be the total time spent in `sync` operations, we derive slowdown using an estimate of the the optimal total latency:

$$\begin{aligned} \text{latency} &= nT_s + \frac{b}{B} \\ \text{slowdown} &= \frac{t}{\text{latency}} \end{aligned} \tag{3.2}$$

To deal with disk caching, buffering, and readahead, we only count as seeks the operations that took longer than a certain threshold, *e.g.*, 5ms. We use similar logic for reads and to estimate the load of each workflow.

Network The load of a workflow on a network link is proportional to the number of bytes transferred by that workflow. We ignore data sent over the loopback interface by checking remote address when the connection is set up, inside our AspectJ instrumentation. We currently do not measure the actual network latency and thus estimate the network slowdown based on its utilization by treating it as a M/M/1 queue. Thus a link with utilization u has slowdown:

$$\text{slowdown} = 1 + \frac{u}{1-u} \tag{3.3}$$

It is feasible to extend Retro by encoding a model of the network (topology, bandwidths, and round trip times), and network flow parameters (source, destination, number of bytes), to estimate the network flow latency with no congestion [196]. Comparing this no-congestion estimate with measured latency could be used to compute network slowdown.

Thread pool The load of a workflow on a thread pool is proportional to the total amount of time it was using threads in this pool. Since we explicitly measure queuing time t_q and service time t_s of a thread pool operation, we can directly compute the slowdown:

$$\text{slowdown} = \frac{t_q + t_s}{t_s} \tag{3.4}$$

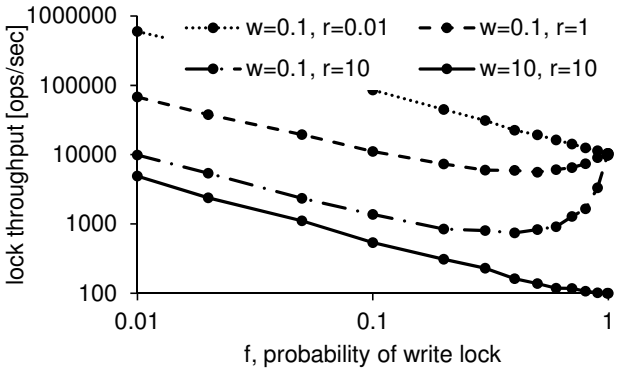


Figure 3.4: The throughput of Java ReentrantReadWriteLock (y-axis) as a function of three parameters: probability of a write lock operation (x-axis), average duration of read and write locks (see legend, time in milliseconds).

Write Locks A write lock behaves similarly to a thread pool with a single thread, and we explicitly measure the queuing time of a lock operation and the time the thread was holding the lock. Slowdown is thus the total latency of lock operation (from requesting the lock until release) divided by the time actually holding the lock.

Read-Write Locks Load of a read-write lock depends on the number of read and write operations, for how long they hold the lock, and the exact lock implementation. While there has been previous work on modeling locks using queues [142, 144, 201], none of them exactly match the `ReentrantReadWriteLock` used in HDFS. Instead, we approximate the capacity or throughput of a lock, $T(f, w, r)$, in a simple benchmark using three workflow parameters: fraction of write locks f, and average duration of write and read locks w and r. See Figure 3.4 for a subset of the measured values; notice that the throughput is nonlinear and non-monotonic. We use trilinear interpolation [143] to predict throughput for values not directly measured. Given a workflow with characteristic (f, w, r) and current lock throughput t, we estimate its load on the lock as $t/T(f, r, w)$. For example, a workflow making 1000 lock requests a second with its estimated max throughput of 5000 operations a second, would have a load of 0.2.

3.4.3 Coordinated throttling

Retro is designed to support multiple scheduling schemes, such as various queue schedulers or priority queues. In the current implementation of Retro, each control point is a *per-workflow distributed token bucket*. Threads can request tokens from the current workflow's token bucket, blocking until available. Queues can delay a request from being dequeued until sufficient tokens are available in the corresponding workflow's bucket. For a particular control point and workflow, a policy can set a rate limit R, which is then split (behind the scenes) across all point instances proportionally to the observed throughput. Retro keeps track of new control point instances coming and going – *e.g.*, mappers starting and finishing – and properly distributes the specified limit across them.

So long as each request executes a bounded amount of work, even using a single control point at the entrance to the system is enough for Retro to enforce usage of individual workflows. However, as described in §3.3.1, requests have to periodically pass through control points to guarantee fast convergence of allocation policies. Even without any control points in the system, each resource reports how many times it has been used by a particular workflow. For example, loading a single HDFS block of 64MB would result in approximately 1000 requests to the disk, each reading 64kB of data. These statistics help developers identify blocks of code where requests execute large amount of work and where adding control points helps break down execution and significantly improves convergence of control policies.

In the Hadoop stack, we added several points: in the HDFS NameNode and HBase RegionServer RPC queues, in the HDFS DataNode block sender and receiver, in the Yarn NodeManager, and in the MapReduce mappers when writing to the local disk. Each of these points has a number of instances equal to the number of processes of the particular type.

Notice that we do not need to throttle directly on resource R to enforce resource limits on R. Assume that a workflow is achieving throughput of N_p at point p and has load L_R on R. By setting a throttling rate of αN_p for all points, we will indirectly control the load on R to αL_R.

3.5 Policies

This section describes three targeted reactive resource-management policies that we used to evaluate Retro. Specifically, these policies enforce fairness on the bottleneck resource (§3.5.1), dominant-resource fairness (§3.5.2), and end-to-end latency SLOs (§3.5.3). All of these polices are system-agnostic, resource-agnostic, and can be concisely stated in a few lines of code. These are not the only policies that could be implemented on top of Retro; in fact, we believe that the Retro abstractions allow developers to write more complex policies that consider a combination of fairness and latency, together with other metrics, such as throughput, workflow priorities, or deadlines.

3.5.1 BFair policy

The BFair policy provides bottleneck fairness [127, 128]; *i.e.*, if a resource is overloaded, the policy reduces the total load on this resource while ensuring max-min fairness for workflows that use this resource. This policy can be used to throttle aggressive workflows or to provide DoS protection. It provides coarse-grained performance isolation, since workflows are guaranteed a fair-share of the bottlenecked resource.

The policy, described in Algorithm 1, first identifies the slowest resource `S` in the system according to the `slowdown` measure (Line 2). Then, the policy runs the max-min fairness algorithm with demands estimated by the current load of workflows (Line 4) and resource capacity estimated by the total demand reduced by $1 - \alpha$ to relieve the bottleneck if any (Line 5).

The policy considers `S` to be bottlenecked if its slowdown is greater than a policy-specific threshold `T`. If this is the

```
1  // identify slowest resource
2  S = r in resources() with max slowdown(r)
3  foreach w in workflows()
4      demand[w] = load(S, w)
5      capacity += (1−α)*demand[w]
6
7  // determine fair allocation
8  fair = MaxMinFairness(capacity, demand)
9
10 // apply fair allocation
11 foreach w in workflows()
12     if (slowdown(S) > T && fair[w] < demand[w])  // throttle
13         factor = fair[w] / demand[w]
14     else                                        // probe for more demand
15         factor = (1 + β)
16
17     foreach p in points()
18         set_rate(p, w, factor*get_rate(p, w))
```

Algorithm 1: BFair policy, see §3.5.1.

case and the fair share `fair[w]` of workflow `w` is smaller than its current load (Line 12), the policy throttles the rate by a factor of `fair[w] / demand[w]`. Here, the policy assumes a linear relationship between throughput at control points and the load on resources. If either the resource is not bottlenecked or if a workflow is not meeting its fair share (Line 14), the policy increases the throttling rate by a factor of $1 + \beta$ to probe for more demand.

Notice that this policy performs *coordinated* throttling of the workflow across all the control points; by reducing the rate proportionally on each point, we quickly reduce the load of the workflow on all resources. Parameters α and β control how aggressively the policy reacts to overloaded resources and underutilized workflows respectively. Notice that this policy will throttle only if there is a bottleneck in the system; we can change the definition of a bottleneck using the parameter `T`.

3.5.2 rDRF policy

Dominant resource fairness (DRF) [128] is a multi-resource fairness algorithm with many desirable properties. The rDRF policy (Algorithm 2) calls the original DRF function at Line 16 which requires the current resource demands and capacities of all resources. In a general distributed system, we cannot directly measure the *actual resource demand* of a workflow, but only its current load on a resource. A workflow might not be able to meet its demand due to bottlenecks in the system.

The rDRF policy overcomes this problem by being reactive: making incremental changes and reacting to how the system responds to these changes. At any instant, the policy conservatively assumes that each workflow can increase its current demand by a factor of α (Line 4). This increased allocation provides room for bottlenecked workflows to increase the load on resources.

Similarly, the policy uses the slowdown measure to estimate capacity. At Line 10, when the current slowdown exceeds

```
1  // estimate resource demands based on measured usage
2  foreach w in workflows()
3      foreach r in resources()
4          demand[r,w] = (1+α)*load(r,w)
5
6  // update capacity estimates based on current slowdown
7  capacities = current capacity estimates
8  foreach r in resources()
9      total_load = Σ_w load(r,w)
10     if(slowdown(r) > T_r)   // slowdown exceeds threshold, reduce estimate
11         capacities[r] = min(capacities[r], total_load);
12     else                    // probe for more capacity
13         capacities[r] = max((1+β)*capacities[r], total_load);
14
15 // determine fair allocation
16 share = DRF(demand, cap)
17
18 // apply fair allocation
19 foreach w in workflows()
20     if (share[w] < 1)
21         foreach p in points()
22             set_rate(p, w, share[w]*get_rate(p,w))
```

Algorithm 2: rDRF policy, see §3.5.2.

```
1  // determine high-priority workflow that is missing latency target
2  foreach w in H
3      miss(w) = latency(w) / target_latency(w)
4  h = w in H with max miss(w)
5
6  // compute low-priority workflow gradients
7  foreach l in L
8      g[l] = Σ_r (latency(h,r) * log(slowdown(r)) * load(r,l) / Σ_w load(r,w))
9
10 // normalize gradients to use as weights
11 foreach l in L
12     g[l] /= ∑_k g[k]
13
14 // throttle or relax low-priority workflows
15 foreach l in L
16     if(miss(h) > 1)  // throttle
17         factor = 1-α*(miss(h)-1)*g[l]
18     else             // relax
19         factor = (1 + β)
20
21     foreach p in points()
22         set_rate(p, l, factor*get_rate(p, l))
```

Algorithm 3: LatencySLO policy, see §3.5.3.

a resource-specific threshold, the policy reduces its capacity estimate to the current load. On the other hand, if the slowdown is within the threshold (Line 12) and the current capacity estimate is lower than the current load, the policy increases the capacity estimate by a factor of β to probe for more capacity.

Given estimates of demand and capacity, the DRF() function returns `share[w]`, the fraction of `w`'s demand that was allocated based on dominant-resource fairness. If `share[w]`< 1, we throttle `w` at each point `p` proportionally to its current throughput at `p`.

3.5.3 LatencySLO policy

In the LatencySLO policy, we have a set of high-priority workflows H with a specified target latency SLO (service-level objective). Let L (low-priority) be the remaining workflows. The goal of the policy is to achieve the highest throughput for L, while meeting the latency targets for H. We assume the system has enough capacity to meet the SLOs for H in the absence of the workflows L; in other words, it is not necessary to throttle H. To maximize throughput, we want to throttle workflows in L as little as possible; *e.g.*, if a workflow in L is not using an overloaded resource, it should not be throttled.

Consider a workflow h in H that is missing its target latency. If multiple such workflows exist, the policy choses the one with the maximum `miss` ratio (Line 4). Let t_w be the current request rate of workflow `w` and consider a possible change of this rate to $t_w \times f_w$. The resulting latency l_h of h is some (nonlinear) function of the relative workflow rates f_w of all workflows. The LatencySLO computes an approximate gradient of l_h with respect to f_w and uses the gradient to move the throttling rates in the right direction. Based on the system response, the policy repeats this process until all latency targets are met.

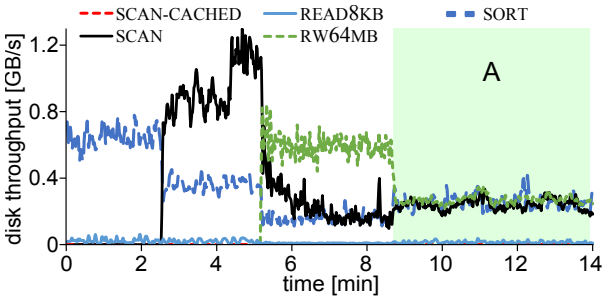
We derive an approximation of l_h which results in an intuitive throttling policy. Consider a resource r with a current slowdown of S_r, load $D_{w,r}$ for workflow w, and total load $D_r = \sum_w D_{w,r}$. If $L_{h,r}$ is the current latency of h at r, the baseline latency is $\frac{L_{h,r}}{S_r}$ when there is no load at r, by the definition of slowdown. We model the latency of h at r, $l_{h,r}$ as an exponential function of the load d_r that satisfies the current ($d_r = D_r$) and baseline ($d_r = 0$) latencies, and obtain $l_{h,r} = L_{h,r} \times S_r^{\frac{d_r}{D_r - 1}}$. Finally, we model the latency of h as $l_h = \sum_r l_{h,r}$ as the sum of latencies across all resources in the system.

Assuming that a fractional change in a workflow's request rate results in the same fractional change in its load on the resources, we have $d_r = \sum_w D_{w,r} \times f_w$. The gradient of $l_{h,r}$ with respect to f_w at $d_r = D_r$ is $\frac{\partial l_{h,r}}{\partial f_w} = \frac{L_{h,r} \times \log S_r \times D_{w,r}}{D_r}$. This is a very intuitive result: the impact of workflow w on the latency of h is high if it has a high resource share, $\frac{D_{w,r}}{D_r}$, on a resource with high slowdown, $\log S_r$, and where workflow h spends a lot of time, $L_{h,r}$.

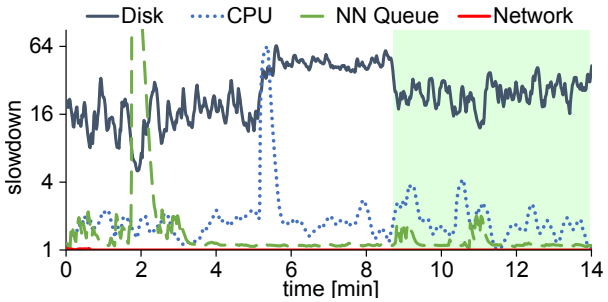
Algorithm 3 uses this formula for the gradient calculation (Line 8). The policy throttles workflows in L based on the normalized gradients after dampening by a factor α to ensure that the policy only takes small steps. If all workflows in H meet their latency guarantees, the policy uses this opportunity to relax the throttling by a factor β.

3.6 Evaluation

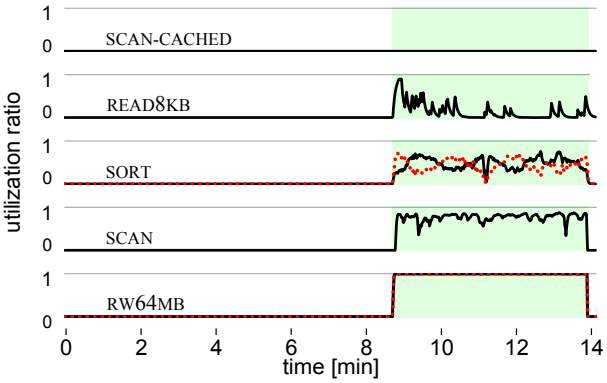
In this section we evaluate Retro in the context of the Hadoop stack. We have instrumented five open-source systems – HDFS, Yarn, MapReduce, HBase, and ZooKeeper – that are widely used in production today. We use a wide variety of workflows, which are based on real-world traces, widely-used benchmarks, and other workloads known to cause resource overload in production systems.



(a) Disk throughput achieved by each workflow.



(b) Slowdown for four resources.



(c) Sparklines illustrating workflow utilizations, *i.e.* their achieved workload throughput relative to their allocated rates. 1 means that the workflow is being actively rate-limited; 0 means the workflow is not rate-limited. SORT plots utilization at two control points (black: DataNode BlockSender; dashed red: Mapper output); also RW64MB (black: DataNode BlockSender; dashed red: DataNode BlockReceiver).

Figure 3.5: BFAIR policy as described in §3.6.1. BFAIR is enabled in phase A with overload threshold T=25.

Our evaluation shows that Retro addresses the challenges in §3.2 when applied simultaneously to all these stack components. In particular, we show that Retro:

- applies coordinated throttling to achieve bottleneck and dominant resource fairness (§3.6.1 and §3.6.3);
- applies policies to application-level resources, resources shared between multiple processes, and resources with multiple instances across the cluster;
- guarantees end-to-end latency in the face of workloads contending on different resources, uniformly for client and system maintenance workflows (§3.6.2);
- is scalable and has very low developer and execution overhead (§3.6.4);
- throttles efficiently: it correctly detects bottlenecked resources and applies *targeted throttling* to the relevant workflows and control points.

We do not directly compare to other policies, since to our knowledge, no previous systems offer this rich source of per-workflow and per-resource data. Many of previous policies, such as Cake [237], could be directly implemented on top of Retro.

3.6.1 BFAIR in Hadoop stack

In Figure 3.5, we demonstrate the BFAIR policy successfully throttling aggressive workflows without negatively affecting the throughput of other workflows. The three *major* workflows are: SORT, a MapReduce sort job; RW64MB, 100 HDFS

clients reading and writing 64MB files with a 50/50 split; and SCAN, 100 HBase clients scanning large tables. These workflows bottleneck on the disk on the worker machines. The two *minor* workflows are: READ8KB, 32 clients reading 8kB files from HDFS; and SCAN-CACHED, 32 clients scanning tables in HBase that are mostly *cached* in the RegionServers. We perform this experiment on a 32-node deployment of Windows Azure virtual machines; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZooKeeper, and HBase RegionServer, the other thirty are used as Hadoop workers. Each VM is a Standard_A4 instance with 8 cores, 14GB RAM and a 600GB data disk, connected by a 1Gbps network.

At the beginning of the experiment, we start READ8KB, SCAN-CACHED, and SORT together, and delay start of SCAN and RW64MB. Figure 3.5a shows the disk throughput achieved by each workflow; notice how the throughput changes as different workflows start, for example, throughput of SORT drops from 750MB/sec to 100MB/sec. Figure 3.5b shows the slowdown of a few different resources. Disk is the only constantly overloaded resource, reaching slowdown of up to 60. While slowdown of other resources also occasionally spikes, this happens only due to workload burstiness. In Figure 3.5c, we show sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point. A ratio of 1 means that the workflow is being actively rate-limited; a ratio of 0 means that the workflow is never rate-limited. For SORT, we show ratios at two control points: the DN BlockSender (black, used by mapper to read data from the DN) and mapper output (dashed red, used by mapper to write its output to local disk). For RW64MB, we show ratios at two control points: the DN BlockSender (black, used to read data from HDFS DNs) and the DN BlockReceiver (dashed red, used to write data to HDFS DNs).

In phase A we enable the BFAIR with overload threshold T=25. Quickly, the disk throughput of the three major workflows equalizes at about 300MB/sec, thus achieving fairness on the bottlenecked resource. Also, the disk slowdown fluctuates at around 25 (navy blue line in the slowdown graph) because the policy starts throttling the major workflows.

The utilization ratio sparklines provide further insight. RW64MB is the most aggressive workflow and consequently it is fully throttled (ratio of 1) at all of the control points. While not as aggressive, SCAN is also throttled though less. Depending on the phase of the map-reduce computation, we throttle SORT while reading input (black) and/or when writing output (red dashed). Finally, as expected, the two minor workflows are not throttled as much, or at all, because the fairness allocates their full demand. Furthermore, SCAN-CACHED is completely unthrottled as it has no disk utilization.

These results highlight how Retro enables coordinated and targeted throttling of workloads. No other system we are aware of would achieve these results, as Retro coordinates the same resource through different control points – for example, disk is controlled not only by HDFS block transfer (used by SCAN, RW64MB, READ8KB and the job input to SORT), but also by the SORT mapper output that accesses disk directly, bypassing HDFS. Retro only throttles the relevant workloads, leaving the small read and scan workloads mostly alone.

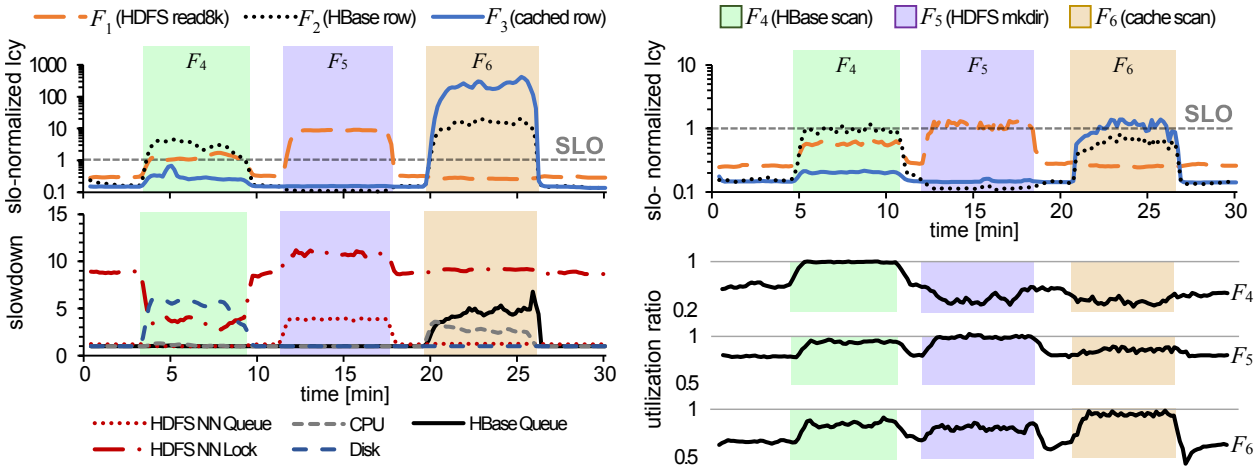


Figure 3.6: LatencySLO policy as described in §3.6.2. Top-left figure shows high priority workflow latencies without LatencySLO. Bottom-left figure shows resource slowdown during experiment. Top-right figure shows high priority workflow latencies with LatencySLO. Bottom-right sparklines show control point utilizations for background workflows.

3.6.2 LatencySLO

We demonstrate that the LatencySLO policy can enforce a) end-to-end latency SLOs across multiple workflows and systems, and b) SLOs for both front-end clients and background tasks. We perform these experiments on an 8-node cluster; one node runs the Retro controller, one node runs HDFS NameNode, Yarn, ZK, and HBase Master, the other 6 are used as Hadoop workers and HBase RegionServers.

Enforcing multiple guarantees In this experiment we simultaneously enforce SLOs in HBase and HDFS for three high priority workflows with intermittently aggressive background workflows. The three high priority workflows are: F_1 randomly reads 8kB from HDFS with 500ms SLO, F_2 randomly reads 1 row from a small table cached by HBase with 25ms SLO, and F_3 randomly reads 1 row from a large HBase table with 250ms SLO. The background workflows are: F_4 submits 400-row HBase table scans, F_5 creates directories in HDFS, and F_6 submits 400-row HBase table scans of a cached HBase table.

Figure 3.6(top-left) demonstrates the request latencies of the three high priority workflows, normalized to their SLOs. During each of the three phases of the experiment, a background workflow temporarily increases its request rate, affecting the latency of the high priority workflows. In the first stage, F_4 increases its load and F_1 and F_2 miss their SLO. In the second stage, F_5 increases its load and F_1 misses its SLO by a factor of 10. In the last stage, F_6 increases its load and F_2 and F_3 miss their SLOs by factors of 10 and 500 respectively. Figure 3.6(bottom-left), shows the slowdown of different resources as the experiment progresses: at first F_4 table scans cause disk slowdown, then F_5 causes HDFS NameNode lock and NameNode queue slowdown, and finally F_6 causes CPU and HBase queue slowdown as its data is cached.

We repeat the experiment using LatencySLO to enforce the SLOs of F_1, F_2 and F_3. Figure 3.6(top-right) shows that the policy successfully maintains the SLOs by throttling the background workflows at a number of control points

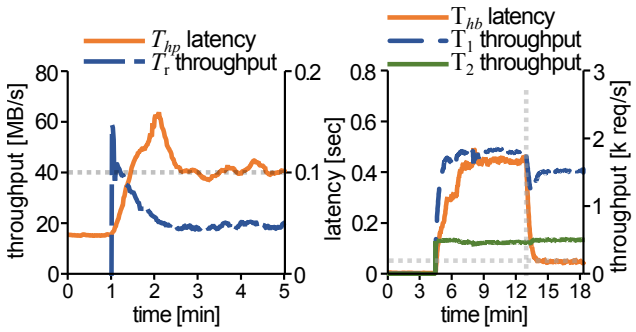


Figure 3.7: LatencySLO rate-limits replication to enforce a 100ms SLO for T_{hp} (left). LatencySLO enforces a 50ms latency for heartbeats (right).

within HDFS and HBase. Figure 3.6(bottom-right) shows the sparklines of the workflow *utilization ratios* – the achieved throughput relative to the allocated rate at a particular control point, similar to Figure 3.5. We see that LatencySLO only rate-limits the background workflows during their specific overload phases.

These results highlight how LatencySLO selectively throttles workloads based on their contribution to the SLO violation. Retro can enforce SLOs for multiple workflows across software and hardware resources simultaneously.

Background workflows Thanks to the workflow abstraction, LatencySLO is equally applicable to providing guarantees for high priority background tasks, such as heartbeats, or to protecting high priority workflows from aggressive background tasks such as data replication.

Figure 3.7 (right) demonstrates the effect of two workflows T_1 and T_2 on the latency of datanode heartbeats, T_{hb}. The heartbeat latency increases from 4ms to about 450ms when T_1 and T_2 start renaming files and listing directories, respectively, causing increased load the HDFS namesystem lock. Whilst T_{hb} and T_2 only require read locks, T_1 requires write locks to update the filesystem, thus blocking heartbeats. When we start SLO enforcement at t=13, the policy identifies T_1 as the cause of slowdown, throttles it at the NameNode RPC queue, and achieves the heartbeat SLO.

Figure 3.7 (left), LatencySLO rate-limits low-priority background replication T_r, to provide guaranteed latency to high priority workflow T_{hp} submitting 8kB read requests with 100ms SLO. At t=1, we manually trigger replication of a large number of HDFS blocks; subsequently, LatencySLO rate-limits T_r. High-priority replication (single remaining replica) could use a separate workflow ID to avoid throttling.

3.6.3 RDRF in HDFS

To demonstrate RDRF (Figure 3.8), we run an experiment with two workflows – READ4M with 50 clients reading 4MB files, and SORT with 5 clients listing 1000 files in a directory – accessing the HDFS cluster remotely sharing a 1Gbps network link. The dominant resource for READ4M is disk and for SORT it is the network, since it is reading large amounts of data from the memory of the NameNode.

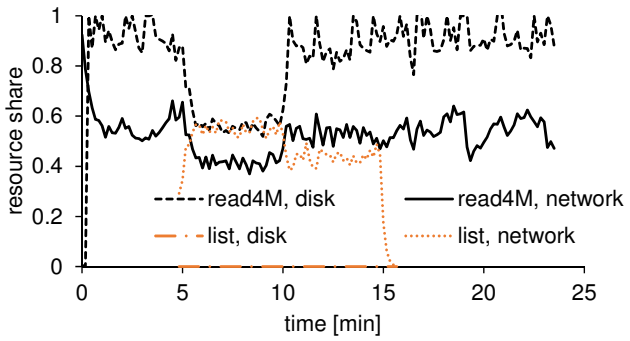


Figure 3.8: Resource share for experiment described in §3.6.3.

We start READ4M at t=0 and add SORT at t=5, with sharing weights of 1. Between time 5 and 10, RDRF throttles READ4M to achieve equal dominant shares across both of these workflows (60% on disk and network). After increasing the weight of READ4M to 2 at t=10, the dominant shares change to 80% and 40%, respectively.

Despite knowing neither the demands of each workflow, nor the capacity of each resource, RDRF successfully allocates each workflow the fair share of its dominant resource. The experiment demonstrates how slowdown is viable as a proxy for resource capacity, and coupled with reactive policies, enables us to overcome some limitations of an existing resource fairness technique.

3.6.4 Overhead and scalability of Retro

Retro propagates a workflow ID (3 bytes) along the execution path of a request, incurring up to 80ns of overhead (see Figure 3.2) to serialize and deserialize when making network calls. The overhead to record a single resource operation is approximately 340ns, which includes intercepting the thread, recording timing, CPU cycle count (before and after the operation), and operation latency, and aggregating these into a per-workflow report.

To estimate the impact of Retro on throughput and end-to-end latency, we benchmark HDFS and HDFS instrumented with Retro using requests derived from the HDFS NNBench benchmark. See Figure 3.9 for throughput and end-to-end latency for five requests types. *Open* opens a file for reading; *Read* reads 8kB of data from a file; *Create* creates a file for writing; *Rename* renames an existing file and *Delete* deletes the file from the name system (and triggers an asynchronous block delete). Of the request types, *Read* is a DataNode operation and the others are NameNode operations. In all cases, latency increases by approximately 1-2%, and throughput drops by a similar 1-2%. Variance in latency and throughput increases slightly in HDFS instrumented with Retro. These overheads could be further significantly reduced by *sampling*, *i.e.*, tracing only a subset of requests or operations.

We evaluate Retro's ability to scale beyond the cluster sizes presented thus far with an 200-VM experiment on Windows Azure (Standard_A2 instances). Figure 3.10 shows slowdown and aggregate disk throughput for four workflows when BFAIR is activated (at t=1.5) and per-workflow weights are adjusted (at t=4). Each workflow ran a mix of 64MB

Operation	Latency
Deserialize metadata	80ns
Read active metadata	9ns
Serialize metadata	46ns
Record use one resource operation	342ns

Table 3.2: Costs of Retro instrumentation

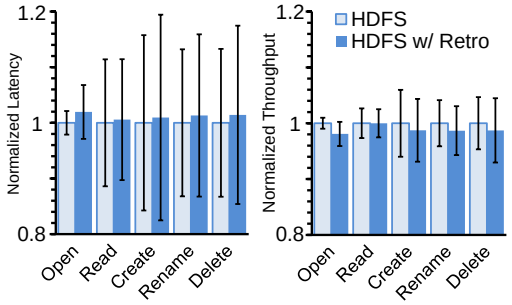


Figure 3.9: Normalized latency (left) and throughput (right) for HDFS NameNode benchmark operations along with error bars showing one standard deviation.

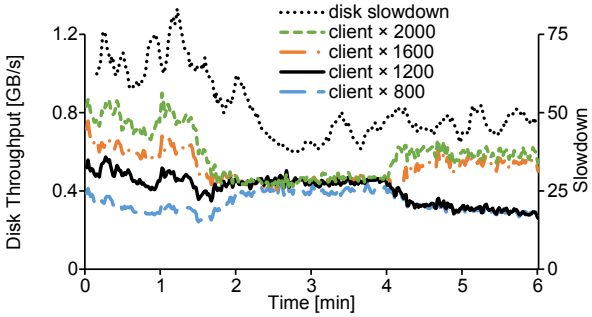


Figure 3.10: Retro's BFair policy on a 200-node cluster with four workflows and overloaded disks. BFair is enabled at t=1.5 with a target slowdown of 50; client weights are adjusted at t=4.

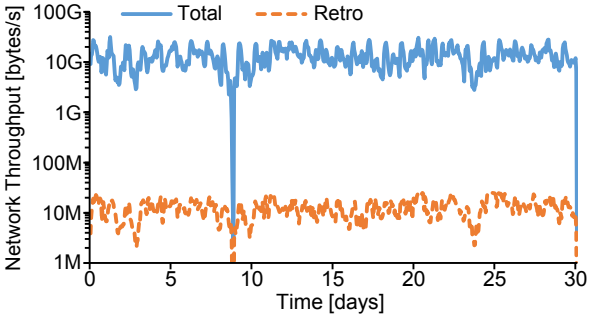


Figure 3.11: Total network throughput for a several-hundred node production Hadoop cluster and network throughput of Retro, from 1 month of traces. Retro's bandwidth requirements are on average 0.1% of the total throughput.

HDFS reads and writes, with 800, 1200, 1600, and 2000 closed-loop clients respectively. Before the policy is activated we observe the expected imbalance in disk throughput caused by the differing number of clients in each workflow. When the policy is activated at t=1.5, the workflows quickly converge to an equal share of disk throughput, and the slowdown decreases to the target of 50. At t=4, two of the clients are given a weight of 2 and the policy quickly establishes the new fair share.

We evaluate the scalability of Retro's central controller in terms of its ability to process resource reports. In a benchmark where each report contains resource usage for 1000 workflows, the controller can process on the order of 10,000 reports per second. Assuming 10 resources per machine, the controller could thus support up to 1000 machines. In this setup, each machine would use about 600kB/sec of network bandwidth to send the reports. Figure 3.11 shows calculated network overhead that would be imposed by Retro on a production Hadoop cluster comprising several hundred nodes over a period of a month. We calculate the network traffic that would be generated by Retro based on traces from this production cluster. The figure shows that Retro would account for an average of 0.1% of the network traffic present. Furthermore, since Retro aggregation only computes sums and averages, we can aggregate hierarchically (*e.g.* inside each machine and rack), further reducing the required network bandwidth and thereby supporting much larger deployments.

Whilst Retro requires manual developer intervention to propagate workflow IDs across network boundaries and to

verify correct behavior of Retro's automatic instrumentation, our experience shows that this requires little work. For example, instrumenting each of the five systems required only between 50 and 200 lines of code; for example to handle RPC messages. Instrumenting resource operations happens automatically through AspectJ.

3.7 Discussion

Retro is a framework for implementing resource management policies in multi-tenant distributed systems. Retro tackles important challenges and provides key abstractions that enable a separation between resource-management policies and mechanisms. It requires low developer effort, and is lightweight enough to be run in production. We demonstrate the applicability of Retro to key components of the Hadoop stack and develop and evaluate three targeted and reactive policies for achieving fairness and latency targets. These policies are system-agnostic, resource-agnostic, and uniformly treat all system activities, including background management tasks. To the best of our knowledge, Retro is the first framework to do so. We plan to extend the control points to provide fair scheduling, prioritization, and load balancing.

In Retro, we made the decision to implement both resource measurement and control points at the application level. While applying Retro in the OS, hypervisor, or device driver level could provide more accurate measurements and fine-granularity enforcement, our approach has the advantages of fast and pervasive deployment, and of not requiring specially built OS or drivers (we deployed Retro in both Windows and Linux environments). Retro's promising results indicate that OS's, and distributed systems in general, should provide mechanisms to facilitate the propagation of workflow IDs across their components.

Retro is extensible to handle *custom resources*. For example, in systems with row-level locking we cannot treat each lock/row as an individual resource because the number of resources might be unbounded. Instead, we could define each data partition as a logical resource, which would significantly reduce the number of resources in the system. ZooKeeper uses a custom request processing pipeline, which is not part of Java standard library. We treat ZooKeeper queues as custom resources and estimate their load and slowdown.

The current implementation of Retro has several limitations. First, some resources cannot be automatically *revoked* once a request has obtained them and have to be explicitly released by the system. For example, this applies to memory, sockets, or disk space. A developer could implement application-specific hooks that Retro could use to reclaim resources. Second, because the rates of distributed token buckets are updated only once a second, when workload is very variable, this might reduce the throughput of the system. Using different local schedulers, such as weighted fair queues [216] and reservations [132] would alleviate this problem.

Chapter 4

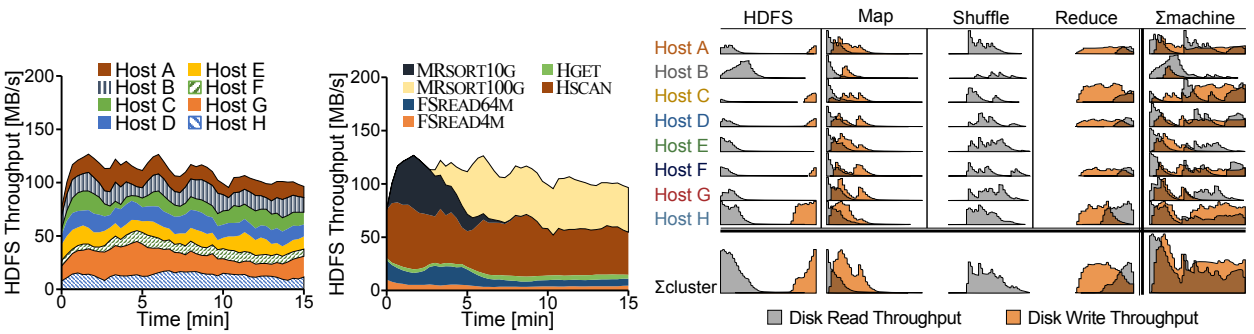
Preliminary Work: Dynamic Causal Monitoring

In this preliminary work I developed Pivot Tracing, a distributed systems monitoring framework. Pivot Tracing gives users, at runtime, the ability to define arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries. Pivot Tracing models the monitoring and tracing of a system as high-level queries over a dynamic dataset of distributed events, and introduces the "happened-before join" operator, $\overrightarrow{\bowtie}$, that enables queries to be contextualized by Lamport's happened-before relation, $\rightarrow$ [155]. Using $\overrightarrow{\bowtie}$, queries can group and filter events based on properties of any events that causally precede them in an execution. To track the happened-before relation between events, Pivot Tracing propagates partial query state along the execution path of each request. This enables inline evaluation of joins during request execution, drastically mitigating query overhead and avoiding the scalability issues of global evaluation.

4.1 Pivot Tracing in Action

In this section we motivate Pivot Tracing with a monitoring task on the Hadoop stack. The goal here is to demonstrate some of what Pivot Tracing can do; details of its design, query language, and implementation are introduced in §4.4, §4.5, and §4.6, respectively.

Suppose we want to apportion the disk bandwidth usage across a cluster of eight machines simultaneously running HBase, Hadoop MapReduce, and direct HDFS clients. §4.7 has an overview of these components, but for now it suffices to know that HBase, a database application, accesses data through HDFS, a distributed file system. MapReduce, in addition to accessing data through HDFS, also accesses the disk directly to perform external sorts and to shuffle data between tasks. We run the following client applications:



(a) HDFS DataNode throughput per machine from instrumented `DataNodeMetrics`.

(b) HDFS DataNode throughput grouped by high-level client application.

(c) Pivot table showing disk read and write sparklines for MRSORT10G. Rows group by host machine; columns group by source process. Bottom row and right column show totals, and bottom-right corner shows grand total.

Figure 4.1: In this example, Pivot Tracing exposes a low-level HDFS metric grouped by client identifiers from other applications. Pivot Tracing can expose arbitrary metrics at one point of the system, while being able to select, filter, and group by events meaningful at other parts of the system, even when crossing component or machine boundaries.

FSREAD4M	Random closed-loop 4MB HDFS reads
FSREAD64M	Random closed-loop 64MB HDFS reads
HGET	10kB row lookups in a large HBase table
HSCAN	4MB table scans of a large HBase table
MRSORT10G	MapReduce sort job on 10GB of input data
MRSORT100G	MapReduce sort job on 100GB of input data

By default, the systems expose a few metrics for disk consumption, such as disk read throughput aggregated by each HDFS DataNode. To reproduce this metric with Pivot Tracing, we define a *tracepoint* for the `DataNodeMetrics` class, in HDFS, to intercept the `incrBytesRead(int delta)` method. A tracepoint is a location in the application source code where instrumentation can run, cf. §4.4. We then run the following query, in Pivot Tracing's LINQ-like query language [167]:

```
Q1: From incr In DataNodeMetrics.incrBytesRead
    GroupBy incr.host
    Select incr.host, SUM(incr.delta)
```

This query causes each machine to aggregate the `delta` argument each time `incrBytesRead` is invoked, grouping by the host name. Each machine reports its local aggregate every second, from which we produce the time series in Figure 4.1a.

Things get more interesting, though, if we wish to measure the HDFS usage of each of our client applications. HDFS only has visibility of its direct clients, and thus an aggregate view of all HBase and all MapReduce clients. At best, applications must estimate throughput client side. With Pivot Tracing, we define tracepoints for the client protocols of HDFS (`DataTransferProtocol`), HBase (`ClientService`), and MapReduce (`ApplicationClientProtocol`), and use the name of the client process as the group by key for the query. Figure 4.1b shows the global HDFS read throughput of each client application, produced by the following query:

```
Q2: From incr In DataNodeMetrics.incrBytesRead
    Join cl In First(ClientProtocols) On cl -> incr
    GroupBy cl.procName
    Select cl.procName, SUM(incr.delta)
```

The -> symbol indicates a happened-before join. Pivot Tracing's implementation will record the process name the first time the request passes through any client protocol method and propagate it along the execution. Then, whenever the execution reaches `incrBytesRead` on a DataNode, Pivot Tracing will emit the bytes read or written, grouped by the recorded name. This query exposes information about client disk throughput that cannot currently be exposed by HDFS.

Figure 4.1c demonstrates the ability for Pivot Tracing to group metrics along arbitrary dimensions. It is generated by two queries similar to Q2 which instrument Java's `FileInput- Stream` and `FileOutputStream`, still joining with the client process name. We show the per-machine, per-application disk read and write throughput of MRSORT10G from the same experiment. This figure resembles a pivot table, where summing across rows yields per-machine totals, summing across columns yields per-system totals, and the bottom right corner shows the global totals. In this example, the client application presents a further dimension along which we could present statistics.

Query Q1 above is processed locally, while query Q2 requires the propagation of information from client processes to the data access points. Pivot Tracing's query optimizer installs dynamic instrumentation where needed, and determines when such propagation must occur to process a query. The out-of-the box metrics provided by HDFS, HBase, and MapReduce cannot provide analyses like those presented here. Simple correlations – such as determining *which* HDFS datanodes were read from by a high-level client application – are not typically possible. Metrics are ad hoc between systems; HDFS sums IO bytes, while HBase exposes operations per second. There is very limited support for cross-tier analysis: MapReduce simply counts global HDFS input and output bytes; HBase does not explicitly relate HDFS metrics to HBase operations.

4.2 Pivot Tracing Overview

Figure 4.2 presents a high-level overview of how Pivot Tracing enables queries such as Q2. We refer to the numbers in the figure (*e.g.*, ①) in our description. Full support for Pivot Tracing in a system requires two basic mechanisms: dynamic code injection and causal metadata propagation. While it is possible to have some of the benefits of Pivot Tracing without one of these (§4.8), for now we assume both are available.

Queries in Pivot Tracing refer to variables exposed by one or more *tracepoints* – places in the system where Pivot Tracing can insert instrumentation. Tracepoint definitions are not part of the system code, but are rather instructions on where and how to change the system to obtain the exported identifiers. Tracepoints in Pivot Tracing are similar to pointcuts from aspect-oriented programming [150], and can refer to arbitrary interface/method signature combinations. Tracepoints are defined by someone with knowledge of the system, maybe a developer or expert operator, and define the

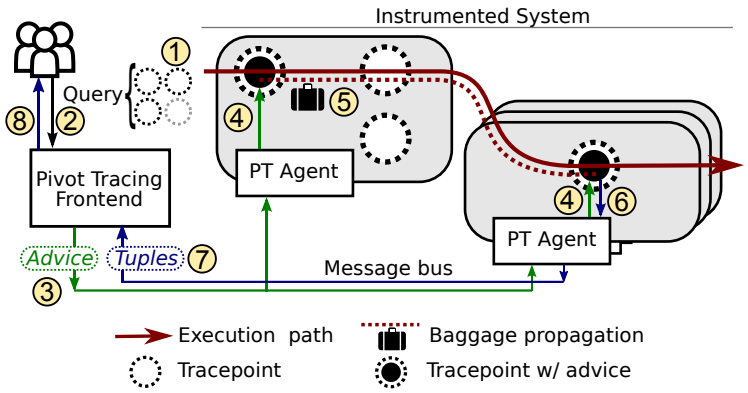


Figure 4.2: Pivot Tracing overview (§4.2)

vocabulary for queries (①). They can be defined and installed at any point in time, and can be shared and disseminated.

Pivot Tracing models system events as tuples of a streaming, distributed dataset. Users submit relational queries over this dataset (②), which get compiled to an intermediate representation called *advice* (③). Advice uses a small instruction set to process queries, and maps directly to code that local Pivot Tracing agents install dynamically at relevant tracepoints (④). Later, requests executing in the system invoke the installed advice each time their execution reaches the tracepoint.

We distinguish Pivot Tracing from prior work by supporting *joins* between events that occur within and across process, machine, and application boundaries. The efficient implementation of the happened before join requires advice in one tracepoint to send information along the execution path to advice in subsequent tracepoints. This is done through a new *baggage* abstraction, which uses causal metadata propagation (⑤). In query Q2, for example, `cl.procName` is packed in the first invocation of the `ClientProtocols` tracepoint, to be accessed when processing the `incrBytesRead` tracepoint.

Advice in some tracepoints also emit tuples (⑥), which get aggregated locally and then finally streamed to the client over a message bus (⑦ and ⑧).

4.3 Monitoring and Troubleshooting Challenges

Pivot Tracing addresses two main challenges in monitoring and troubleshooting. First, when the choice of what to record about an execution is made a priori, there is an inherent tradeoff between recall and overhead. Second, to diagnose many important problems one needs to correlate and integrate data that crosses component, system, and machine boundaries.

One size does not fit all Problems in distributed systems are complex, varied, and unpredictable. By default, the information required to diagnose an issue may not be reported by the system or contained in system logs. Current approaches tie logging and statistics mechanisms into the development path of products, where there is a mismatch between the expectations and incentives of the developer and the needs of operators and users. Panelists at SLAML [99]

discussed the important need to *"close the loop of operations back to developers"*. According to Yuan *et al.* [248], regarding diagnosing failures, *"(…) existing log messages contain too little information. Despite their widespread use in failure diagnosis, it is still rare that log messages are systematically designed to support this function."*

This mismatch can be observed in the many issues raised by users on Apache's issue trackers: to request new metrics [37, 41, 46–48, 64, 74]; to request changes to aggregation methods [49, 63, 66]; and to request new breakdowns of existing metrics [36, 44, 45, 56–59, 62, 63, 70, 72, 76, 79, 89]. Many issues remain unresolved due to developer pushback [57, 62, 72, 74, 79] or inertia [44, 46, 47, 64, 66, 70, 76, 89]. Even simple cases of misconfiguration are frequently unreported by error logs [247].

Eventually, applications may be updated to record more information, but this has effects both in performance and information overload. Users must pay the performance overheads of any systems that are enabled by default, regardless of their utility. For example, HBase SchemaMetrics were introduced to aid developers, but all users of HBase pay the 10% performance overhead they incur [63]. The HBase user guide [67] carries the following warning for users wishing to integrate with Ganglia [166]: *"By default, HBase emits a large number of metrics per region server. Ganglia may have difficulty processing all these metrics. Consider increasing the capacity of the Ganglia server or reducing the number of metrics emitted by HBase."*

The glut of recorded information presents a "needle-in-a-haystack" problem to users [202]; while a system may expose information relevant to a problem, *e.g.* in a log, extracting this information requires system familiarity developed over a long period of time. For example, Mesos cluster state is exposed via a single JSON endpoint and can become massive, even if a client only wants information for a subset of the state [90].

Dynamic instrumentation frameworks such as Fay [122], DTrace [102], and SystemTap [200] address these limitations, by allowing almost arbitrary instrumentation to be installed dynamically at runtime, and have proven extremely useful in the diagnosis of complex and subtle system problems [101]. Because of their side-effect-free nature, however, they are limited in the extent to which probes may share information with each other. In Fay, only probes in the same address space can share information, while in DTrace the scope is limited to a single operating system instance.

Crossing Boundaries This brings us to the second challenge Pivot Tracing addresses. In multi-tenant, multi-application stacks, the root cause and symptoms of an issue may appear in different processes, machines, and application tiers, and may be visible to different users. A user of one application may need to relate information from some other dependent application in order to diagnose problems that span multiple systems. For example, HBASE-4145 [58] outlines how MapReduce lacks the ability to access HBase metrics on a per-task basis, and that the framework only returns aggregates across all tasks. MESOS-1949 [89] outlines how the executors for a task do not propagate failure information, so diagnosis can be difficult if an executor fails. In discussion the developers note: *"The actually interesting / useful information is hidden in one of four or five different places, potentially spread across as many different machines. This leads to unpleasant*

Operation	*Description*	*Example*
From	Use input tuples from a set of tracepoints	`From e In RPCs`
Union ($\bigcup$)	Union events from multiple tracepoints	`From e In DataRPCs, ControlRPCs`
Selection (σ)	Filter only tuples that match a predicate	`Where e.Size < 10`
Rename (ρ)	Create a field using an expression	`Let SizeKB = e.Size / 1000`
Projection (Π)	Restrict tuples to a subset of fields	`Select e.User, e.Host`
Aggregation (A)	Aggregate tuples	`Select SUM(e.Cost)`
GroupBy (G)	Group tuples based on one or more fields	`GroupBy e.User`
GroupBy Aggregation (GA)	Aggregate tuples of a group	`Select e.User, SUM(e.Cost)`
Happened-Before Join ($\overrightarrow{\bowtie}$)	Happened-before join tuples from another query	`Join d In Disk On d->e`
	Happened-before join a subset of tuples	`Join d In MostRecent(Disk) On d->e`

Table 4.1: Operations supported by the Pivot Tracing query language

and repetitive searching through logs looking for a clue to what went wrong. (…) There's a lot of information that is hidden in log files and is very hard to correlate."

Prior research has presented mechanisms to observe or infer the relationship between events (§2.1.2) and studies of logging practices conclude that end-to-end tracing would be helpful in navigating the logging issues they outline [180, 202]. A variety of these mechanisms have also materialized in production systems: for example, Google's Dapper [220], Apache's HTrace [84], Accumulo's Cloudtrace [11], and Twitter's Zipkin [233]. These approaches can obtain richer information about particular executions than component-centric logs or metrics alone, and have found uses in troubleshooting, debugging, performance analysis and anomaly detection, for example. However, most of these systems record or reconstruct traces of execution for offline analysis, and thus share the problems above with the first challenge, concerning what to record.

4.4 Design

We now detail the fundamental concepts and mechanisms behind Pivot Tracing. Pivot Tracing is a dynamic monitoring and tracing framework for distributed systems. At a high level, it aims to enable flexible runtime monitoring by correlating metrics and events from arbitrary points in the system. The challenges outlined in §4.3 motivate the following high-level design goals:

1. Dynamically configure and install monitoring at runtime
2. Low system overhead to enable "always on" monitoring
3. Capture causality between events from multiple processes and applications

4.4.1 Tracepoints

Tracepoints provide the system-level entry point for Pivot Tracing queries. A tracepoint typically corresponds to some event: a client sends a request; a low-level IO operation completes; an external RPC is invoked, etc.

A tracepoint identifies one or more locations in the system code where Pivot Tracing can install and run instrumentation. Tracepoints export named variables that can be accessed by instrumentation. Figure 4.5 shows the specification of one of the tracepoints in `Q2` from §4.1. Besides declared exports, all tracepoints export a few variables by default: host, timestamp, process id, process name, and the tracepoint definition.

Tracepoints are only references to locations where Pivot Tracing can install instrumentation — they are not baked into the system and they do not require a priori modifications. Only at runtime, when a user submits a query, will Pivot Tracing compile and install monitoring code at tracepoints referenced by the query. Subsequently, when requests running in the system reach a tracepoint, any instrumentation configured for that tracepoint will be invoked, generating a tuple with its exported variables. These are then accessible to any instrumentation code installed at the tracepoint.

4.4.2 Query Language

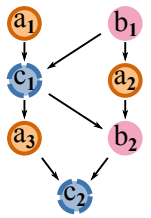
Pivot Tracing enables users to express high-level queries about the variables exported by one or more tracepoints. We abstract tracepoint invocations as streaming datasets of tuples; Pivot Tracing queries are therefore relational queries across the tuples of several such datasets.

To express queries, Pivot Tracing provides a parser for LINQ-like text queries such as those outlined in §4.1. Table 4.1 outlines the query operations supported by Pivot Tracing. Pivot Tracing supports several typical operations including projection (Π), selection (σ), renaming (ρ), grouping (`G`), and aggregation (`A`). Pivot Tracing aggregators include `Count`, `Sum`, `Max`, `Min`, and `Average`. Pivot Tracing also defines the temporal filters `MostRecent`, `MostRecentN`, `First`, and `FirstN`, to take the `1` or `N` most or least recent events. Finally, Pivot Tracing introduces the *happened-before join* query operator ($\overrightarrow{\bowtie}$).

4.4.3 Happened-before Joins

A key contribution of Pivot Tracing is the happened-before join query operator. Happened-before join enables the tuples from two Pivot Tracing queries to be joined based on Lamport's happened before relation, $\rightarrow$ [155]. For events a and b occurring anywhere in the system, we say that a happened before b and write $a \rightarrow b$ if the occurrence of event a causally preceded the occurrence of event b and they occurred as part of the execution of the same request.[1] If a and b are not part of the same execution, then $a \not\rightarrow b$; if the occurrence of a did not lead to the occurrence of b, then $a \not\rightarrow b$ (*e.g.*, they occur in two parallel threads of execution that do not communicate); and if $a \rightarrow b$ then $b \not\rightarrow a$.

[1] This definition does not capture all possible causality, including when events in the processing of one request could influence another, but could be extended if necessary.



(a) A request that triggers tracepoints A, B and C several times during its execution. Each invocation of a tracepoint produces a tuple, *e.g.* tracepoint A generates a_1, a_2, a_3, etc.

Query	*Query Results*
A	a_1, a_2, a_3
A $\overrightarrow{\bowtie}$ B	$a_1 \rightarrow b_2$, $a_2 \rightarrow b_2$
B $\overrightarrow{\bowtie}$ C	$b_1 \rightarrow c_1$, $b_1 \rightarrow c_2$, $b_2 \rightarrow c_2$
(A $\overrightarrow{\bowtie}$ B) $\overrightarrow{\bowtie}$ C	$a_1 \rightarrow b_2 \rightarrow c_2$, $a_2 \rightarrow b_2 \rightarrow c_2$

(b) Queries and the corresponding results for the request. A query for a single tracepoint (*e.g.* A) gives all tuples produced by that tracepoint. A happened-before join between two tracepoints (*e.g.* A $\overrightarrow{\bowtie}$ B) gives all pairs of tuples satisfying that happened-before relationship.

Figure 4.3: An example request execution graph and the results of running queries over that request.

In general, events occurring during a request's execution will form a *directed, acyclic graph* (DAG) under the happened-before relation. Figure 4.3 illustrates an example DAG. Events occurring concurrently in different threads, processes, or machines, do not satisfy the happened-before relation (*e.g.*, a_1 and b_1). However, when there is communication between concurrent components, that communication will establish the happened-before relationship with later events (*e.g.*, $a_1 \rightarrow c_1$ and $b_1 \rightarrow c_1$).

The happened-before join operator enables queries about the relationships between events. For any two queries Q_1 and Q_2, the happened-before join $Q_1 \overrightarrow{\bowtie} Q_2$ produces tuples $t_1 t_2$ for all $t_1 \in Q_1$ and $t_2 \in Q_2$ such that $t_1 \rightarrow t_2$. That is, Q_1 produced t_1 before Q_2 produced tuple t_2 in the execution of the same request. Figure 4.3 shows an example execution triggering tracepoints A, B, and C several times, and outlines the tuples that would be produced for this execution by different queries.

Query `Q2` in §4.1 demonstrates the use of happened-before join. In the query, tuples generated by the disk IO tracepoint `DataNodeMetrics.incrBytesRead` are joined to the first tuple generated by the `ClientProtocols` tracepoint.

Happened-before join substantially improves our ability to perform root cause analysis by giving us visibility into the relationships *between* events in the system. The happened-before relationship is fundamental to a number of prior approaches in root cause analysis (§2.1.2). Pivot Tracing is designed to efficiently support happened-before joins, but does not optimize more general joins such as equijoins ($\bowtie$).

4.4.4 Advice

Pivot Tracing queries compile to an intermediate representation called *advice*. Advice specifies the operations to perform at each tracepoint used in a query, and eventually materializes as monitoring code installed at those tracepoints (§4.6). Advice has several operations for manipulating tuples through the tracepoint-exported variables, and evaluating $\overrightarrow{\bowtie}$ on tuples produced by other advice at prior tracepoints in the execution.

Table 4.2 outlines the advice API. OBSERVE creates a tuple from exported tracepoint variables. UNPACK retrieves

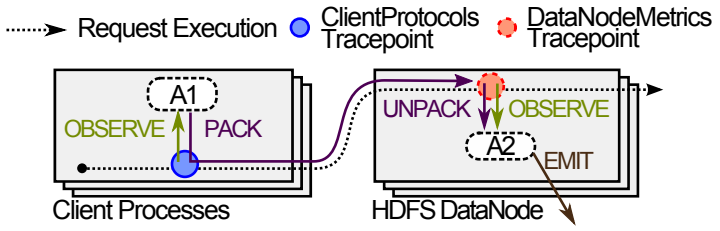
Operation	*Description*
OBSERVE	Construct a tuple from variables exported by a tracepoint
UNPACK	Retrieve one or more tuples from prior advice
FILTER	Evaluate a predicate on all tuples
PACK	Make tuples available for use by later advice
EMIT	Output a tuple for global aggregation

Table 4.2: Primitive operations supported by Pivot Tracing advice for generating and aggregating tuples as defined in §4.4.

```
A1: OBSERVE procName
    PACK-FIRST procName

A2: OBSERVE delta
    UNPACK procName
    EMIT procName, SUM(delta)
```

(a) Advice generated for `Q2` from §4.1. `A1` is dynamically installed at the `ClientProtocols` tracepoint, and `A2` at the `DataNodeMetrics` tracepoint.



(b) When requests pass through `ClientProtocols` they invoke `A1`. `A1` observes and packs `procName` to be carried with the request. When requests subsequently reach `DataNodeMetrics`, `A2` unpacks `procName`, observes `delta`, and emits `(procName, SUM(delta))`

Figure 4.4: Pivot Tracing evaluates query `Q2` from §4.1 by compiling it into two *advice* specifications. Pivot Tracing dynamically installs the advice at the tracepoints referenced in the query.

tuples generated by other advice at other tracepoints prior in the execution. Unpacked tuples can be joined to the observed tuple, *i.e.*, if t_o is observed and t_{u1} and t_{u2} are unpacked, then the resulting tuples are $t_o t_{u1}$ and $t_o t_{u2}$. Tuples created by this advice can be discarded (FILTER), made available to advice at other tracepoints later in the execution (PACK), or output for global aggregation (EMIT). Both PACK and EMIT can group tuples based on matching fields, and perform simple aggregations such as `SUM` and `COUNT`. PACK also has the following special cases: `FIRST` packs the first tuple encountered and ignores subsequent tuples; `RECENT` packs only the most recent tuple, overwriting existing tuples. `FIRSTN` and `RECENTN` generalize this to `N` tuples. The advice API is expressive but restricted enough to provide some safety guarantees. In particular, advice code has no jumps or recursion, and is guaranteed to terminate.

By packing and unpacking tuples into the baggage, advice can join tuples based on the happened-before relation: if we are interested in emitting tuples only when A happened before B, a query can pack tuples at A and unpack tuples at B. Since baggage is explicitly propagated along the execution path of a request, this query directly evaluates the happened-before relation, as follows:

Theorem 1. *Let e_1 be any event, and let t be a tuple observed at that event,* i.e. $t = \text{OBSERVE}(e_1)$. *If we* PACK t *at e_1, then for all other events e_2 we get the following:*

$$t \in \text{UNPACK}(e_2) \iff e_1 \rightarrow e_2$$

Proof. Suppose $t \in \text{UNPACK}(e_2)$. Then the baggage at e_1 was propagated to e_2. By definition, if the baggage at e_1 is propagated to e_2 then e_2 is part of the same execution. Therefore $e_1 \rightarrow e_2$.
Now suppose $e_1 \rightarrow e_2$ and $t \notin \text{UNPACK}(e_2)$. Then the baggage at e_1 is not the same baggage as at e_2. By definition, if e_2 is part of the same execution as e_1 then the baggage at e_1 is propagated to e_2. Therefore e_2 is not part of the same execution so $e_1 \not\rightarrow e_2$. □

Example Figure 4.4 outlines the advice generated for query `Q2` from §4.1, and illustrates how the advice and tracepoints interact with the execution of requests in the system. First, `A1` observes and packs a single valued tuple containing the process name. Then, when execution reaches the `DataNodeMetrics` tracepoint, `A2` unpacks the process name, observes the value of `delta`, then emits a joined tuple. Figure 4.4 shows how this advice and the tracepoints interact with the execution of requests in the system.

Compiling Queries to Advice To compile a query to advice, we instantiate one advice specification for a `From` clause and add an OBSERVE operation for the tracepoint variables used in the query. For each `Join` clause, we add an UNPACK operation for the variables that originate from the joined query. We recursively generate advice for the joined query, and append a PACK operation at the end of its advice for the variables that we unpacked. `Where` directly translates to a FILTER operation. We add an EMIT operation for the output variables of the query, restricted according to any `Select` clause. `Aggregate`, `GroupBy`, and `GroupByAggregate` are all handled by EMIT and PACK. §4.5 outlines several query rewriting optimizations for implementing $\overrightarrow{\bowtie}$.

Installing Advice at Tracepoints Pivot Tracing *weaves* advice into tracepoints by: 1) loading code that implements the advice operations; 2) configuring the tracepoint to execute that code and pass its exported variables; 3) activating the necessary tracepoint at all locations in the system. Figure 4.5 outlines this process of weaving advice for `Q2`.

4.5 Pivot Tracing Optimizations

In this section we outline several optimizations that Pivot Tracing performs in order to support efficient evaluation of happened-before joins.

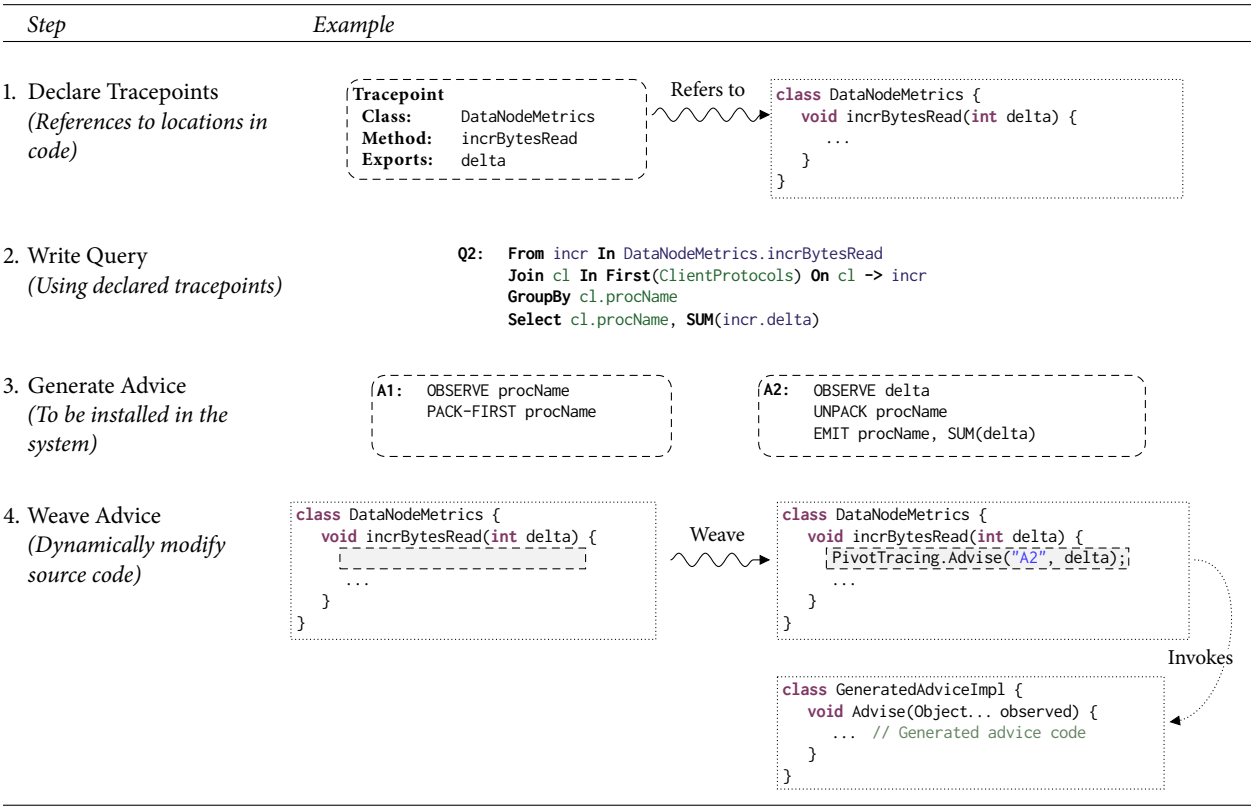


Figure 4.5: Steps to install a Pivot Tracing query. Tracepoints are only references to locations in code, and require no system-level modifications until queries are installed. Step 4 illustrates how we *weave* advice for `Q2` at the `DataNodeMetrics` tracepoint (`Q2` also weaves advice at `ClientProtocols`, not shown). Variables exported by the tracepoint (*i.e.*, `delta`) are passed when the advice is invoked.

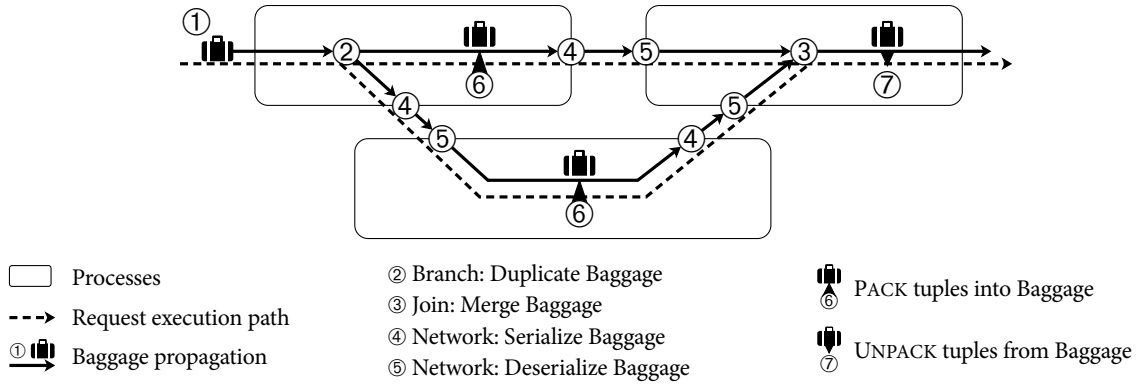


Figure 4.6: To implement the happened-before join, systems propagate baggage (①) along the execution path of requests. Baggage is duplicated when requests split into concurrent branches (②); merged when concurrent branches join (③); and included in all inter-process communication (④ ⑤). Pivot Tracing queries PACK tuples into the baggage at some tracepoints (⑥) and UNPACK tuples at other tracepoints (⑦).

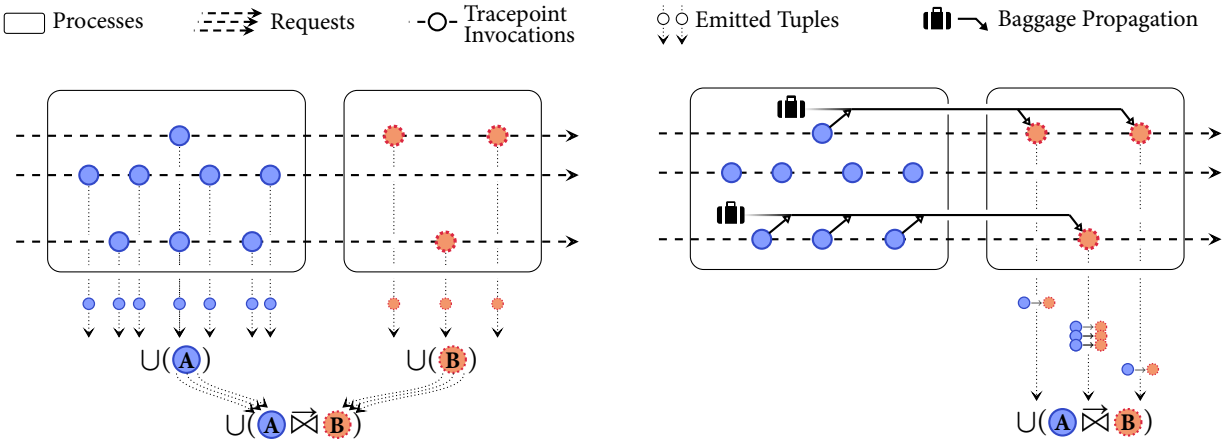
4.5.1 Baggage

The naïve evaluation strategy for happened-before join is that of an equijoin ($\bowtie$) or θ-join ($\bowtie_\theta$ [203]), requiring tuples to be aggregated globally across the cluster prior to evaluating the join. Temporal joins as implemented by Magpie [95], for example, are expensive because they implement this evaluation strategy (§2.1.2). Figure 4.7a illustrates this approach for happened-before join.

Instead of a naïve global join, Pivot Tracing enables inexpensive happened-before joins by providing the *baggage* abstraction. Baggage is a per-request container for tuples that is propagated alongside a request as it traverses thread, application and machine boundaries. Figure 4.6 illustrates baggage propagation for a request. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches; merged when concurrent branches join; and included with all of the request's inter-process and inter-thread communication. Pivot Tracing advice uses the PACK and UNPACK operations to store and retrieve tuples from the current request's baggage.

Baggage explicitly follows the execution path of each request, and thereby observes all happened-before relationships of a request while it is executing. Tuples that are packed during a request's execution will follow the request from that point onward. Consequently, the presence of the tuples in baggage later during the request's execution imply a happened-before relationship.

Baggage is a generalization of end-to-end metadata propagation techniques outlined in prior work such as X-Trace [126] and Dapper [220]. Using baggage, Pivot Tracing efficiently evaluates happened-before joins *in situ* during the execution of a request. Figure 4.7b shows the optimized query evaluation strategy to evaluate joins in-place during request execution.



(a) Unoptimized query with all tuples collected centrally across the cluster prior to evaluation of $\overrightarrow{\bowtie}$.

(b) Optimized query that propagates tuples in baggage, enabling inline evaluation of $\overrightarrow{\bowtie}$.

Figure 4.7: Illustration comparing the optimized and unoptimized evaluation of A $\overrightarrow{\bowtie}$ B. The optimized query evaluates $\overrightarrow{\bowtie}$ inline by propagating tuples in baggage, avoiding a costly global aggregation.

4.5.2 Local Tuple Aggregation

One metric to assess the cost of a Pivot Tracing query is the number of tuples emitted for global aggregation. Although baggage carries partial query state at a per-request granularity, once tuples have been emitted for global aggregation, Pivot Tracing collects and aggregates these tuples across all requests to produce the final query results. To reduce this cost, Pivot Tracing performs intermediate aggregation for queries containing `Aggregate` or `GroupByAggregate`. Pivot Tracing aggregates the emitted tuples within each process and reports results globally at a regular interval, *e.g.*, once per second. Process-level aggregation substantially reduces traffic for emitted tuples; Q2 from §4.1 is reduced from approximately 600 tuples per second to 6 tuples per second from each DataNode.

4.5.3 Optimizing Happened-Before Joins

A second cost metric for Pivot Tracing queries is the number of tuples packed during a request's execution and carried within the request's baggage. Pivot Tracing rewrites queries to minimize the number of tuples packed. Pivot Tracing pushes projection, selection, and aggregation terms as close as possible to source tracepoints. In Fay [122] the authors outlined query optimizations for merging streams of tuples, enabled because projection, selection, and aggregation are distributive. These optimizations also apply to Pivot Tracing and reduce the number of tuples emitted for global aggregation. To reduce the number of tuples transported in the baggage, Pivot Tracing adds further optimizations for happened-before joins, outlined in Table 4.3.

Query	*Optimized Query*
$\Pi_{p,q}(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q})$
$\sigma_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\sigma_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q}$
$\sigma_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathrm{P} \overrightarrow{\bowtie} \sigma_q(\mathrm{Q})$
$\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$

Query	*Optimized Query*
$\mathtt{GA}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_p\mathtt{Combine}_p(\mathtt{GA}_p(\mathrm{P}) \overrightarrow{\bowtie} \mathrm{Q})$
$\mathtt{GA}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_q\mathtt{Combine}_p(\mathrm{P} \overrightarrow{\bowtie} \mathtt{GA}_q(\mathrm{Q}))$
$\mathtt{G}_p\mathtt{A}_q(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_p\mathtt{Combine}_q(\Pi_p(\mathrm{P}) \overrightarrow{\bowtie} \mathtt{A}_q(\mathrm{Q}))$
$\mathtt{G}_q\mathtt{A}_p(\mathrm{P} \overrightarrow{\bowtie} \mathrm{Q})$	$\mathtt{G}_q\mathtt{Combine}_p(\mathtt{A}_p(\mathrm{P}) \overrightarrow{\bowtie} \Pi_q(\mathrm{Q}))$

Table 4.3: Query rewrite rules for happened-before join between two queries P and Q. Optimizations push operators as close as possible to source tuples, thereby reducing the tuples that must be propagated in baggage from P to Q. `Combine` refers to an aggregator's combiner function (*e.g.*, for `Count`, the combiner is `Sum`). See Table 4.1 for descriptions of query operators.

4.5.4 Cost of Baggage Propagation

Pivot Tracing does not inherently bound the number of packed tuples and potentially accumulates a new tuple for every tracepoint invocation. However, we liken this to database queries that inherently risk a full table scan – our optimizations mean that in practice, this is an unlikely event. Several of Pivot Tracing's aggregation operators explicitly restrict the number of propagated tuples and in our experience, queries only end up propagating aggregations, most-recent, or first tuples.

In cases where too many tuples are packed in the baggage, Pivot Tracing could revert to an alternative query plan, where all tuples are emitted instead of packed, and the baggage size is kept constant by storing only enough information to reconstruct the causality, *a la* X-Trace [126], Stardust [232], or Dapper [220]. To estimate the overhead of queries, Pivot Tracing can execute a modified version of the query to count tuples rather than aggregate them explicitly. This would provide live analysis similar to "explain" queries in the database domain.

4.6 Implementation

We have implemented a Pivot Tracing prototype in Java and applied Pivot Tracing to several open-source systems from the Hadoop ecosystem. Section §4.7 outlines our instrumentation of these systems. In this section, we describe the implementation of our prototype.

We opted to implement and evaluate Pivot Tracing in Java in order to make use of several existing open-source distributed systems written in this language. However, the components of Pivot Tracing generalize and are not restricted to Java – a query could span multiple systems written in different programming languages.

4.6.1 Pivot Tracing Agent

A Pivot Tracing agent thread runs in every Pivot Tracing-enabled process and awaits instruction via central pub/sub server to weave advice to tracepoints. Tuples emitted by advice are accumulated by the local Pivot Tracing agent, which performs partial aggregation of tuples according to their source query. Agents publish partial query results at a configurable interval

Table 4.4: Baggage API for Pivot Tracing Java implementation. PACK operations store tuples in the baggage. API methods are static and only allow interaction with the current execution's baggage.

Method	*Description*
`pack(q, t...)`	Pack tuples into the baggage for a query
`unpack(q)`	Retrieve all tuples for a query
`serialize()`	Serialize the baggage to bytes
`deserialize(b)`	Set the baggage by deserializing from bytes
`split()`	Split the baggage for a branching execution
`join(b1, b2)`	Merge baggage from two joining executions

– by default, one second.

4.6.2 Dynamic Instrumentation

Our prototype weaves advice at runtime, providing dynamic instrumentation similar to that of DTrace [102] and Fay [122]. Java version 1.5 onwards supports dynamic method body rewriting via the `java.lang.instrument` package. The Pivot Tracing agent programmatically rewrites and reloads class bytecode from within the process using Javassist [108].

We can define new tracepoints at runtime and dynamically weave and unweave advice. To weave advice, we rewrite method bodies to add advice invocations at the locations defined by the tracepoint (cf. Figure 4.5). Our prototype supports tracepoints at the entry, exit, or exceptional return of any method. Tracepoints can also be inserted at specific line numbers.

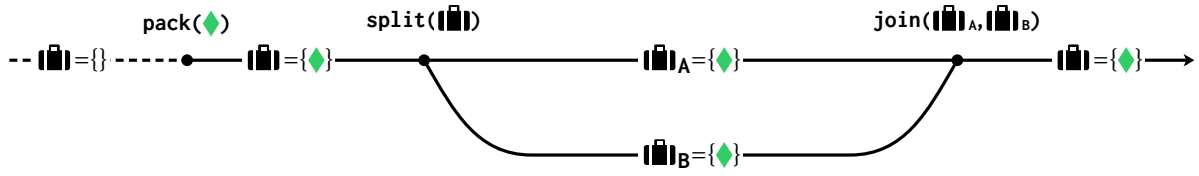
To define a tracepoint, users specify a class name, method name, method signature and weave location. Pivot Tracing also supports pattern matching, for example, all methods of an interface on a class. This feature is modeled after *pointcuts* from AspectJ [149]. Pivot Tracing supports instrumenting privileged classes (*e.g.*, `FileInputStream` in §4.1) by providing an optional agent that can be placed on Java's boot classpath.

Pivot Tracing only makes system modifications when advice is woven into a tracepoint, so inactive tracepoints incur no overhead. Executions that do not trigger the tracepoint are unaffected by Pivot Tracing. Pivot Tracing has a zero-probe effect: methods are unmodified by default, so tracepoints impose truly zero overhead until advice is woven into them.

4.6.3 Baggage

We provide an implementation of Baggage that stores per-request instances in thread-local variables. At the beginning of a request, we instantiate empty baggage in the thread-local variable; at the end of the request, we clear the baggage from the thread-local variable. The baggage API (Table 4.4) can get or set tuples for a query and at any point in time baggage can be retrieved for propagation to another thread or serialization onto the network. To support multiple queries simultaneously, queries are assigned unique IDs and tuples are packed and unpacked based on this ID.

Baggage is lazily serialized and deserialized using protocol buffers [235]. This minimizes the overhead of propagating



(a) One tuple (◆) is packed prior to the execution branching. When the execution splits, the tuple exists in both 🧳A and 🧳B. When the execution joins, the tuple must only be retained once, because it only captures one happened-before relationship.

(b) Baggage is empty when the execution branches. Both branches of execution encounter distinct events that pack a tuple; however, the tuples happen to have the same value (both ◆). When the execution joins, the two identical tuples must both be retained, because they capture different happened-before relationships.

Figure 4.8: An execution that branches then joins. Baggage is split into 🧳A and 🧳B, then later joined. In both examples, 🧳A and 🧳B contain the same tuples; however, the merge result is different in each case, because the tuples represent different happened-before relationships. Baggage must be consistently joined in order to correctly reflect the happened-before relationship represented by the tuples (cf. §4.6.5).

baggage through applications that do not actively participate in a query, since baggage is only deserialized when an application attempts to pack or unpack tuples. Serialization costs are only incurred for modified baggage at network or application boundaries.

Pivot Tracing relies on developers to implement Baggage propagation when a request crosses thread, process, or asynchronous execution boundaries. In our experience (§4.7) this entails adding a baggage field to existing application-level request contexts and RPC headers.

4.6.4 Materializing Advice

Tracepoints with woven advice invoke the `PivotTracing.Advise` method (cf. Figure 4.5), passing tracepoint variables as arguments. Tuples are implemented as `Object` arrays and there is a straightforward translation from advice to implementation: OBSERVE constructs a tuple from the advice arguments; UNPACK and PACK interact with the Baggage API; FILTER discards tuples that do not match a predicate; and EMIT forwards tuples to the process-local aggregator.

4.6.5 Baggage Consistency

In order to preserve the happened-before relation correctly within a request, Pivot Tracing must handle executions that branch and rejoin, as illustrated in Figure 4.6. Each branch of the request's execution maintains its own baggage instance. To propagate baggage, instances are copied when executions split into concurrent branches; merged when concurrent branches join; and included with all of the request's inter-process and inter-thread communication. Tuples packed by one execution branch are not visible to any other branch until the branches rejoin or communicate.

In executions that arbitrarily branch and join, we must be careful not to inadvertently duplicate tuples when merging baggage instances from concurrent execution branches. It is possible for both baggage instances to contain the same tuple derived from the same source event, as illustrated in Figure 4.8a. In this case our merge function must not naïvely duplicate the tuple – we must ensure that the output baggage only contains the tuple once, to correctly reflect that there is only one happened-before relationship. On the other hand, it is also possible for each execution branch to independently pack tuples with the same values, as illustrated in Figure 4.8b. In this case, we expect the merged baggage to contain both tuples, because they represent independent events and we must preserve both happened-before relationships.

To correctly preserve happened-before relationships for executions that arbitrarily branch and join, we implement baggage as a *set*: when a request branches, the baggage tuples are simply duplicated for each branch; when two or more branches join, we perform a *set union* on the tuples present in each baggage instance. To disambiguate between the two cases illustrated in Figure 4.8, we append a unique identifier to each tuple when it is initially packed. Consequently, when merging two baggage instances, set union will not duplicate tuples if they are the same tuple derived from the same source event; conversely, if two distinct events produced tuples with the same value, they will be differentiated by different IDs.

We extend this concept further to handle the query optimization rules described in §4.5.3, which push projection, selection, and aggregation terms as close as possible to source tuples. For some queries, these optimizations lead to advice that directly aggregates tuples in baggage, for example, by summing values immediately as tuples are packed, and propagating only the sum in baggage. However, for the same reason as illustrated in Figure 4.8, it is insufficient to only propagate a running total in baggage, as it can lead to double counting when merging baggage instances. Instead, we require some additional causality information to determine how to merge two sums.

To handle this, we extend the previously described tuple ID scheme. Internal to a baggage instance, we maintain one or more *bags*, each of which has a unique bag ID and sequence number. We store tuples, aggregations, etc. within each bag naïvely without any additional embellishments. Each branch of execution considers one of the bags to be the 'active' bag, and it packs and aggregates tuples into only that bag. After an execution branches, one of the branches continues using the previous active bag and increments the bag's sequence number; the other branch lazily instantiates a new bag the next time it needs to pack tuples. When multiple branches join, the bag with the highest sequence number for each

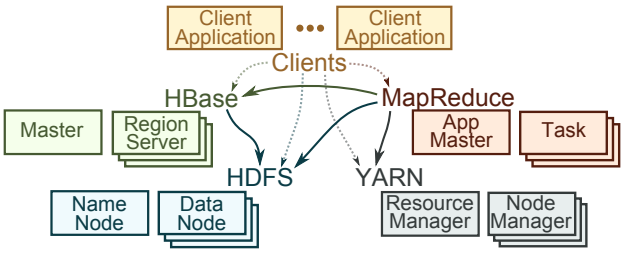


Figure 4.9: Interactions between systems. Each system comprises several processes on potentially many machines. Typical deployments often co-locate processes from several applications, *e.g.* DataNode, NodeManager, Task and RegionServer processes.

bag ID is retained.

Under this scheme a baggage instance resembles a version vector [199] with a variable number of components, for which bags correspond to components. In our current implementation, we randomly generate a bag ID when creating a bag. Previously, we generated bag IDs by using interval tree clocks [5], which provide a mechanism for splitting IDs when executions branch, and recombining IDs when branches rejoin. However, we found that this scheme introduced undesirable overhead, because it required maintaining and splitting IDs even if nothing was propagated in the baggage. By contrast, random IDs have no such overhead.

4.7 Evaluation

In this section we evaluate Pivot Tracing in the context of the Hadoop stack. We have instrumented four open-source systems – HDFS, HBase, Hadoop MapReduce, and YARN – that are widely used in production today. We present several case studies where we used Pivot Tracing to successfully diagnose root causes, including real-world issues we encountered in our cluster and experiments presented in prior work [156, 239]. Our evaluation shows that Pivot Tracing addresses the challenges in §4.3 when applied to these stack components. In particular, we show that Pivot Tracing:

- is dynamic and extensible to new kinds of analysis (§4.7.2)
- is scalable and has low developer and execution overhead (§4.7.3)
- enables cross-tier analysis between any inter-operating applications (§4.1, §4.7.2)
- captures event causality to successfully diagnose root causes (§4.7.1, §4.7.2)
- enables insightful analysis with even a very small number of tracepoints (§4.7.1)

Hadoop Overview We first give a high-level overview of Hadoop, before describing the necessary modifications to enable Pivot Tracing. Figure 4.9 shows the relevant components of the Hadoop stack.

HDFS [218] is a distributed file system that consists of several DataNodes that store replicated file blocks and a NameNode that manages the filesystem metadata.

HBase [39] is a non-relational database modeled after Google's Bigtable [105] that runs on top of HDFS and comprises

a Master and several RegionServer processes.

Hadoop MapReduce is an implementation of the MapReduce programming model [114] for large-scale data processing, that uses YARN containers to run map and reduce tasks. Each job runs an ApplicationMaster and several MapTask and ReduceTask containers.

YARN [236] is a container manager to run user-provided processes across the cluster. NodeManager processes run on each machine to manage local containers, and a centralized ResourceManager manages the overall cluster state and requests from users.

Hadoop Instrumentation In order to support Pivot Tracing in these systems, we made one-time modifications to propagate baggage along the execution path of requests. As described in §4.6 our prototype uses a thread-local variable to store baggage during execution, so the only required system modifications are to set and unset baggage at execution boundaries. To propagate baggage across remote procedure calls, we manually extended the protocol definitions of the systems. To propagate baggage across execution boundaries within individual processes we implemented AspectJ [149] instrumentation to automatically modify common interfaces (`Thread`, `Runnable`, `Callable`, and `Queue`). Each system only required between 50 and 200 lines of manual code modification. Once modified, these systems could support arbitrary Pivot Tracing queries without further modification.

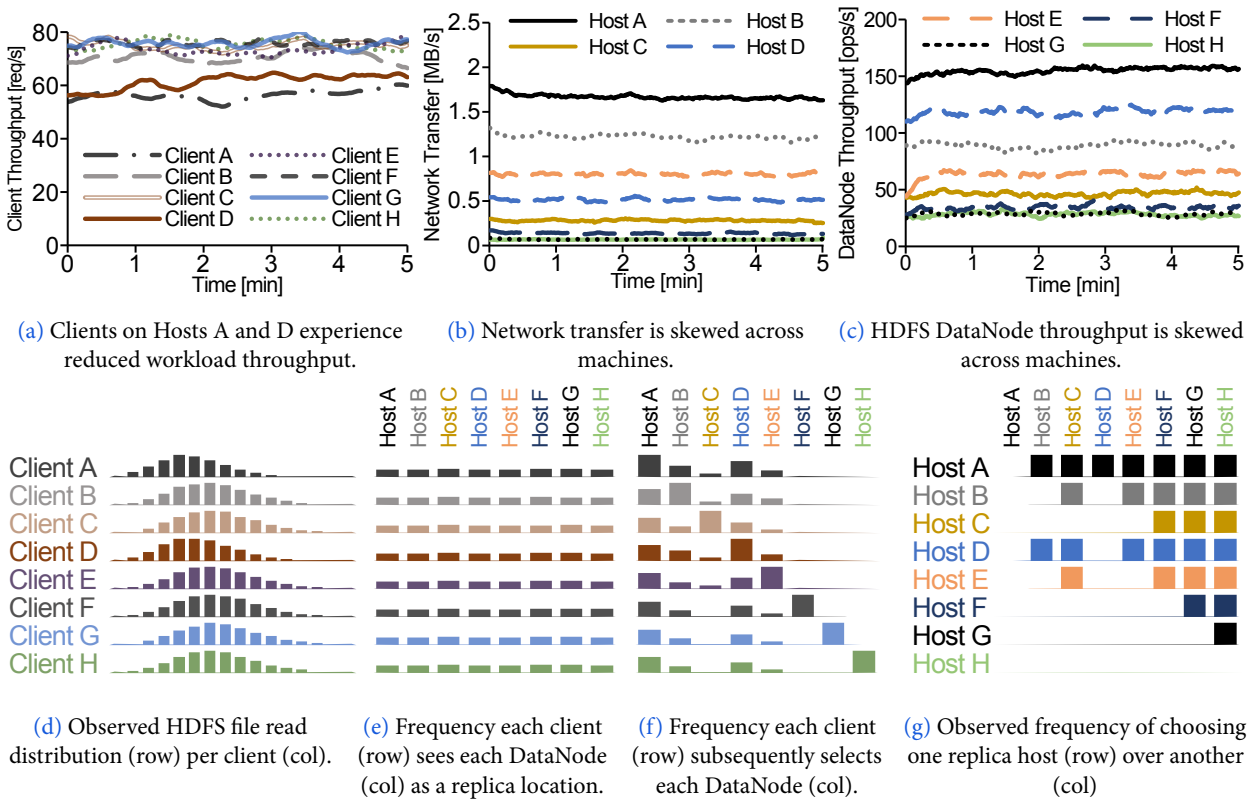
Our queries used tracepoints from both client and server RPC protocol implementations of the HDFS DataNode `DataTransferProtocol` and NameNode `ClientProtocol`. We also used tracepoints for piggybacking off existing metric collection mechanisms in each instrumented system, such as `DataNodeMetrics` and `RPCMetrics` in HDFS and `MetricsRegionServer` in HBase.

4.7.1 Case Study: HDFS Replica Selection Bug

In this section we describe our discovery of a replica selection bug in HDFS that resulted in uneven distribution of load to replicas. After identifying the bug, we found that it had been recently reported and subsequently fixed in an upcoming HDFS version [75].

HDFS provides file redundancy by decomposing files into blocks and replicating each block onto several machines (typically 3). A client can read any replica of a block and does so by first contacting the NameNode to find replica hosts (`GetBlockLocations`), then selecting the closest replica as follows: 1) read a local replica; 2) read a rack-local replica; 3) select a replica at random. We discovered a bug whereby rack-local replica selection always follows a global static ordering due to two conflicting behaviors: the HDFS client does not randomly select between replicas; and the HDFS NameNode does not randomize rack-local replicas returned to the client. The bug results in heavy load on the some hosts and near zero load on others.

In this scenario we ran 96 stress test clients on an HDFS cluster of 8 DataNodes and 1 NameNode. Each machine has



(a) Clients on Hosts A and D experience reduced workload throughput.

(b) Network transfer is skewed across machines.

(c) HDFS DataNode throughput is skewed across machines.

(d) Observed HDFS file read distribution (row) per client (col).

(e) Frequency each client (row) sees each DataNode (col) as a replica location.

(f) Frequency each client (row) subsequently selects each DataNode (col).

(g) Observed frequency of choosing one replica host (row) over another (col)

Figure 4.10: Pivot Tracing query results leading to our discovery of HDFS-6268 [75]. Faulty replica selection logic led clients to prioritize the replicas hosted by particular DataNodes (§4.7.1).

identical hardware specifications; 8 cores, 16GB RAM, and a 1Gbit network interface. On each host, we ran a process called StressTest that used an HDFS client to perform closed-loop random 8kB reads from a dataset of 10,000 128MB files with a replication factor of 3.

Our investigation of the bug began when we noticed that the stress test clients on hosts A and D had consistently lower request throughput than clients on other hosts, shown in Figure 4.10a, despite identical machine specifications and setup. We first checked machine level resource utilization on each host, which indicated substantial variation in the network throughput (Figure 4.10b). We began our diagnosis with Pivot Tracing by first checking to see whether an imbalance in HDFS load was causing the variation in network throughput. The following query installs advice at a DataNode tracepoint that is invoked by each incoming RPC:

```
Q3: From dnop In DataNode.DataTransferProtocol
    GroupBy dnop.host
    Select dnop.host, COUNT
```

Figure 4.10c plots the results of this query, showing the HDFS request throughput on each DataNode. It shows that DataNodes on hosts A and D in particular have substantially higher request throughput than others – host A has on average 150 ops/sec, while host H has only 25 ops/sec. This behavior was unexpected given that our stress test clients are supposedly reading files uniformly at random. Our next query installs advice in the stress test clients and on the HDFS NameNode, to correlate each read request with the client that issued it:

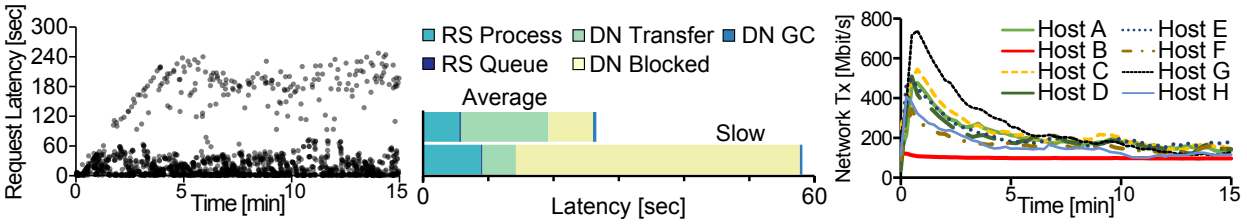
```
Q4: From getloc In NameNode.GetBlockLocations
    Join st In StressTestClient.DoNextOp On st -> getloc
    GroupBy st.host, getloc.src
    Select st.host, getloc.src, COUNT
```

This query counts the number of times each client reads each file. In Figure 4.10d we plot the distribution of counts over a 5 minute period for clients from each host. The distributions all fit a normal distribution and indicate that all of the clients are reading files uniformly at random. The distribution of reads from clients on A and D are skewed left, consistent with their overall lower read throughput.

Having confirmed the expected behavior of our stress test clients, we next checked to see whether the skewed datanode throughput was simply a result of skewed block placement across datanodes:

```
Q5: From getloc In NameNode.GetBlockLocations
    Join st In StressTestClient.DoNextOp On st -> getloc
    GroupBy st.host, getloc.replicas
    Select st.host, getloc.replicas, COUNT
```

This query measures the frequency that each DataNode is hosting a replica for files being read. Figure 4.10e shows that, for each client, replicas are near-uniformly distributed across DataNodes in the cluster. These results indicate that clients have an equal opportunity to read replicas from each DataNode, yet, our measurements in 4.10c clearly show that they do not. To gain more insight into this inconsistency, our next query relates the results from 4.10e and 4.10c:



(a) HBase Request Latencies (b) Request latency decomposition. (c) Per-Machine Network Throughput

Figure 4.11: (a) Observed request latencies for a closed-loop HBase workload experiencing occasional end-to-end latency spikes; (b) Average time in each component on average (top), and for slow requests (bottom, end-to-end latency > 30s); (c) Per-machine network throughput – a faulty network cable has downgraded Host B's link speed to 100Mbit, affecting entire cluster throughput.

```
Q6: From DNop In DataNode.DataTransferProtocol
    Join st In StressTestClient.DoNextOp On st -> DNop
    GroupBy st.host, DNop.host
    Select st.host, DNop.host, COUNT
```

This query measures the frequency that each client selects each DataNode for reading a replica. We plot the results in Figure 4.10f and see that the clients are clearly favoring particular DataNodes. The strong diagonal is consistent with HDFS client preference for locally-hosted replicas (39% of the time in this case). However, the expected behavior when there is not a local replica is to select a rack-local replica uniformly at random; clearly these results suggest that this was not happening.

Our final diagnosis steps were as follows. First, we checked to see *which* replica was selected by HDFS clients from the locations returned by the NameNode. We found that clients always selected the first location returned by the NameNode. Second, we measured the conditional probabilities that DataNodes precede each other in the locations returned by the NameNode. We issued the following query for the latter:

```
Q7: From DNop In DataNode.DataTransferProtocol
    Join getloc In NameNode.GetBlockLocations On getloc -> DNop
    Join st In StressTestClient.DoNextOp On st -> getloc
    Where st.host != DNop.host
    GroupBy DNop.host, getloc.replicas
    Select DNop.host, getloc.replicas, COUNT
```

This query correlates the DataNode that is selected with the other DataNodes also hosting a replica. We remove the interference from locally-hosted replicas by *filtering* only the requests that do a non-local read. Figure 4.10g shows that host A was *always* selected when it hosted a replica; host D was always selected except if host A was also a replica, and so on.

At this point in our analysis, we concluded that this behavior was quite likely to be a bug in HDFS. HDFS clients did not randomly select between replicas, and the HDFS NameNode did not randomize the rack-local replicas. We checked Apache's issue tracker and found that the bug had been recently reported and fixed in an upcoming version of HDFS [75].

4.7.2 Diagnosing End-to-End Latency

Pivot Tracing can express queries about the time spent by a request across the components it traverses using the built-in `time` variable exported by each tracepoint. Advice can pack the timestamp of any event then unpack it at a subsequent event, enabling comparison of timestamps between events. The following example query measures the latency between receiving a request and sending a response:

```
Q8: From response In SendResponse
    Join request In MostRecent(ReceiveRequest) On request -> response
    Select response.time - request.time
```

When evaluating this query, **`MostRecent`** ensures we select only the most recent preceding ReceiveRequest event whenever SendResponse occurs. We can use latency measurement in more complicated queries. The following example query measures the average request latency experienced by Hadoop jobs:

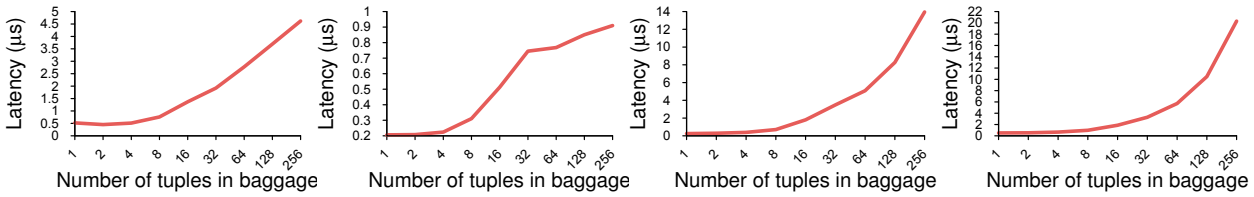
```
Q9: From job In JobCompletionEvents
    Join latencyMeasurement In Q8 On latencyMeasurement -> end
    Select job.id, AVERAGE(latencyMeasurement)
```

A query can measure latency in several components and propagate measurements in the baggage, reminiscent of transaction tracking in Timecard [204] and transactional profiling in Whodunit [103]. For example, during the development of Pivot Tracing we encountered an instance of network limplock [119, 156], whereby a faulty network cable caused a network link downgrade from 1Gbit to 100Mbit. One HBase workload in particular would experience latency spikes in the requests hitting this bottleneck link (Figure 4.11a). To diagnose the issue, we decomposed requests into their per-component latency and compared anomalous requests (> 30s end-to-end latency) to the average case (Figure 4.11b). This enabled us to identify the bottleneck source as time spent blocked on the network in the HDFS DataNode on Host B. We measured the latency and throughput experienced by all workloads at this component and were able to identify the uncharacteristically low throughput of Host B's network link (Figure 4.11c).

We have also replicated results in end-to-end latency diagnosis in the following other cases: to diagnose rogue garbage collection in HBase RegionServers as described in [239]; and to diagnose an overloaded HDFS NameNode due to exclusive write locking as described in [162].

4.7.3 Overheads of Pivot Tracing

Baggage By default, Pivot Tracing propagates an empty baggage with a serialized size of 0 bytes. In the worst case Pivot Tracing may need to pack an unbounded number of tuples in the baggage, one for each tracepoint invoked. However, the optimizations in §4.5 reduce the number of propagated tuples to 1 for `Aggregate`, 1 for `Recent`, n for `GroupBy` with n



(a) Pack 1 tuple (b) Unpack all tuples (c) Serialize baggage (d) Deserialize baggage

Figure 4.12: Latency micro-benchmark results for packing, unpacking, serializing, and deserializing randomly-generated 8-byte tuples.

groups, and `N` for `RecentN`. Of the queries presented in this chapter, `Q7` propagates the largest baggage containing the stress test hostname and the location of all 3 file replicas (4 tuples, ≈137 bytes per request).

The size of serialized baggage is approximately linear in the number of packed tuples. The overhead to pack and unpack tuples from the baggage varies with the size of the baggage – Figure 4.12 gives micro-benchmark results measuring the overhead of baggage API calls.

Application-level Overhead To estimate the impact of Pivot Tracing on application-level throughput and latency, we ran benchmarks from HiBench [139], YCSB [111], and HDFS DFSIO and NNBench benchmarks. Many of these benchmarks bottleneck on network or disk and we noticed no significant performance change with Pivot Tracing enabled.

To measure the effect of Pivot Tracing on CPU bound requests, we stress tested HDFS using requests derived from the HDFS NNBench benchmark: Read8k reads 8kB from a file; Open opens a file for reading; Create creates a file for writing; Rename renames an existing file. Read8kB is a DataNode operation and the others are NameNode operations. We compared the end-to-end latency of requests in unmodified HDFS to HDFS modified in the following ways: 1) with Pivot Tracing enabled; 2) propagating baggage containing one tuple but no advice installed; 3) propagating baggage containing 60 tuples (≈1kB) but no advice installed; 4) with the advice from queries in §4.7.1 installed; 5) with the advice from queries in §4.7.2 installed.

Table 4.5 shows that the application-level overhead with Pivot Tracing enabled is at most 0.3%. This overhead includes the costs of baggage propagation within HDFS, baggage serialization in RPC calls, and to run Java in debugging mode. The most noticeable overheads are incurred when propagating 60 tuples in the baggage, incurring 15.9% overhead for Open. Since this is a short CPU-bound request (involving a single read-only lookup), 16% is within reasonable expectations. Rename does not trigger any advice for the queries from §4.7.1, but does trigger advice for the queries from §4.7.2. Overheads of 0.3% and 5.5% respectively reflect this difference.

Dynamic Instrumentation Some Java Virtual Machines (JVMs) only support the HotSwap feature when debugging mode is enabled, which in turn disables some compiler optimizations. For practical purposes, however, HotSpot JVM's full-speed debugging is sufficiently optimized that it is possible to run with debugging support always enabled [192]. Our HDFS throughput experiments above measured only a small overhead between debugging enabled and disabled.

	Benchmark			
	READ8K	OPEN	CREATE	RENAME
Unmodified System	0%	0%	0%	0%
Pivot Tracing Enabled	0.3%	0.3%	<0.1%	0.2%
Baggage – 1 Tuple	0.8%	0.4%	0.6%	0.8%
Baggage – 60 Tuples	0.8%	15.9%	8.6%	4.1%
Queries – §4.7.1	1.5%	4.0%	6.0%	0.3%
Queries – §4.7.2	1.9%	14.3%	8.2%	5.5%

Table 4.5: Latency overheads for HDFS stress test with Pivot Tracing enabled, baggage propagation enabled, and full queries enabled, as described in §4.7.3

Reloading a class with woven advice has a one-time cost of approximately 100ms, depending on the size of the class being reloaded.

4.8 Discussion

Pivot Tracing is the first monitoring system to combine dynamic instrumentation and causal tracing. Its novel happened-before join operator fundamentally increases the expressive power of dynamic instrumentation and the applicability of causal tracing. Pivot Tracing enables cross-tier analysis between any inter-operating applications, with low execution overhead. Ultimately, its power lies in the uniform and ubiquitous way in which it integrates monitoring of a heterogeneous distributed system.

Despite the advantages over logs and metrics for troubleshooting (§4.3), Pivot Tracing is not meant to replace all functions of logs, such as security auditing, forensics, or debugging [180].

Pivot Tracing is designed to have similar per-query overheads to the metrics currently exposed by systems today. It is feasible for a system to have several Pivot Tracing queries on by default; these could be sensible defaults provided by developers, or custom queries installed by users to address their specific needs. We leave it to future work to explore the use of Pivot Tracing for automatic problem detection and exploration.

Dynamic instrumentation is not a requirement to utilize Pivot Tracing. By default, a system could hard-code a set of predefined tracepoints. Without dynamic instrumentation the user is restricted to those tracepoints; adding new tracepoints remains tied to the development and build cycles of the system. Inactive tracepoints would also incur at least the cost of a conditional check, instead of our current zero cost. Similarly, a system that does not implement baggage can still utilize the other mechanisms of Pivot Tracing; in such a case the system resembles DTrace [102] or Fay [122]. Alternatively, kernel-level execution context propagation [104, 198, 205] can provide language-independent access to baggage variables.

While users are restricted to advice comprised of Pivot Tracing primitives, Pivot Tracing does not guarantee that its queries will be side-effect free, due to the way exported variables from tracepoints are currently defined. We can enforce

that only trusted administrators define tracepoints and require that advice be signed for installation, but a comprehensive security analysis, including complete sanitization of tracepoint code is beyond the scope of this work.

The experiments included in this thesis evaluate Pivot Tracing on an 8-node cluster. However, initial runs of the instrumented systems on a 200-node cluster with constant-size baggage being propagated showed negligible performance impact. It is ongoing work to evaluate the scalability of Pivot Tracing to larger clusters and more complex queries. Sampling at the advice level is a further method of reducing overhead which we plan to investigate.

We opted to implement Pivot Tracing in Java in order to easily instrument several popular open-source distributed systems written in this language. However, the components of Pivot Tracing generalize and are not restricted to Java – a query can span multiple systems written in different programming languages due to Pivot Tracing's platform-independent baggage format and restricted set of advice operations. In particular, it would be an interesting exercise to integrate the happened-before join with Fay or DTrace.

Chapter 5

Proposed Work

5.1 Abstractions for Tracing Distributed Systems

Retro and Pivot Tracing demonstrate how context propagation is a powerful runtime mechanism for capturing causal relationships between prior events on the execution path of requests. Context propagation is also an important component in a range of further distributed system monitoring, diagnosis, and debugging tasks (cf. §2.1.2).[1]

In practice, however, organizations report drawn-out struggles to deploy tools like these, because of the high developer cost associated with *instrumentation*: the small but non-trivial modifications to system source code necessary for propagating the tool's context across request execution boundaries, *e.g.* to serialize contexts to RPC headers, to maintain them in thread-local storage, etc. Researchers and practitioners consistently describe instrumentation as the most significant obstacle to deploying a tracing tool because *all* system components must be instrumented to participate in context propagation: on the one hand, developers must be intimately familiar with the system and its libraries, such as event loops, futures, and queues, to determine boundaries for propagating contexts; on the other hand, it requires coherent choices and participation across all system components, even though developers of different systems and components are often isolated from one another and make incompatible or conflicting choices about which tracing tools to deploy. Finally, most tracing tools, including Retro and Pivot Tracing, lack abstractions to enable other tools to reuse or extend existing instrumentation, so achieving end-to-end visibility is hard because it requires coherent tracing tool choices across system components at development time.

[1] In this thesis I use the term *tracing tools* to refer to tools that use context propagation for some cross-component monitoring, diagnosis, or debugging task. Prior work has also used the term *meta-applications* for such tools [104]

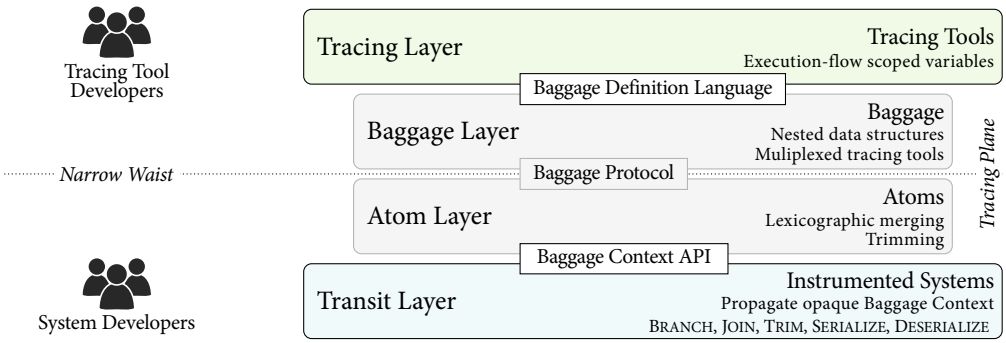


Figure 5.1: Tracing Plane layered design.

	Concerns	*Tasks*	*Abstractions*
Tracing Tool Developers (*e.g.* Zipkin, Pivot Tracing)	Tracing tool semantics, information communicated	Tracing tool logic: define context using BDL; use execution-path scoped variables to implement tracing logic (e.g., spans, security, resource accounting, etc.)	Execution-flow scoped variables
Tracing Plane Developers (This proposal)	Context behavior under concurrent execution (*e.g.* avoid double-counting)	Tracing plane internals: BDL compiler, general-purpose underlying context format, context multiplexing and trimming	Baggage; Atoms
System Developers (*e.g.* HDFS, Spark, etc.)	Execution boundaries, concurrency events, RPCs, queues, threadpools, etc.	Instrumentation: Propagate baggage across boundaries; demarcate execution start, end, branch, join; serialize, deserialize	Baggage context: opaque context propagation

Table 5.1: The Tracing Plane provides abstractions to separate the concerns of different developers.

5.2 Decoupling Tracing Tools

Many of these problems stem from a tight coupling between tracing tools and the instrumentation needed to propagate their contexts, which yield brittle deployments and repeated developer effort. I propose instead that system instrumentation should be a one-time task, reusable, independent of any tracing tool, and that developing, deploying, and updating tracing tools should be possible without having to revisit or consider the underlying context propagation mechanisms. To achieve this I propose two key abstractions that, together, separate the concerns of tracing tool developers from those of system developers, and enable reusable general-purpose context propagation.

First, tracing tool developers should implement contexts using the abstraction of *execution-flow scoped variables*, which I liken to thread-local storage but dynamically scoped to end-to-end executions instead of threads. These variables are grouped into what I term *baggage*, which is a generalization of the key-value pairs used by Pivot Tracing.

Second, system developers doing instrumentation should propagate opaque contexts, which I term *baggage contexts*. Baggage contexts hide their underlying context format from system developers and standardize instrumentation on a simple set of five propagation primitives, thus avoiding development-time decisions about which tracing tools to support.

To encapsulate these high level abstractions, I propose a layered architecture called the *Tracing Plane*, illustrated in Figure 5.1. The Tracing Plane is an abstraction layering that separates the concerns of system developers from those of tracing tool developers, described in Table 5.1.

5.3 Tracing Plane External Interfaces

The bottommost layer of the Tracing Plane, the transit layer, is the proposed entry point for system developers. Its main concern is the pervasive propagation of opaque context metadata, *i.e.* baggage contexts. System developers are responsible for propagating the opaque baggage context objects alongside the execution. The transit layer aims at producing reusable instrumentation, that is only done once, independently of tracing tools. To this end, it has two requirements. First, that baggage contexts be opaque, which decouples the instrumentation from the details of any specific tracing tool. Second, that the instrumentation be pervasive: with opaque contexts, we have no idea which causal relationships a tracing tool might want to preserve; consequently, it is necessary to instrument all execution boundaries.

The topmost layer of the Tracing Plane, the tracing layer, will greatly simplify the development and deployment of tracing tools. Tracing tools will be able to share the underlying Tracing Plane infrastructure while ignoring details of concurrency and propagation. The abstraction provided to the tracing tool developer is that of *execution-flow scoped variables*, which are analogous to thread-local variables, but with a scope that follows the execution even across system components. In prior work, tracing tools have used a wide range of data types, from simple primitives to data structures like sets, maps, aggregations, graphs, and more [110, 130, 164, 228]. Since executions can arbitrarily branch and join, these data types also need well-defined behavior for those boundaries, *i.e.* to duplicate and merge instances. This complicates the implementation of many data types, and in the general case contexts are analogous to state-based replicated objects [212]. For example, to avoid subtle concurrency issues such as double-counting, Pivot Tracing's `SUM` aggregation mirrors the PN-Counter CRDT [213]. To hide these subtleties from tracing tool developers, I propose that the tracing layer is exposed by means of a rich library of concurrent data types. A tracing tool will define its data types through an interface definition language called BDL, or Baggage Definition Language. BDL will compile to object representations that use the services of the Tracing Plane's inner layers to implement data types and nested data structures, all with well-defined branch and join semantics.

5.4 Tracing Plane Internals

The intermediary layers of the Tracing Plane – the atom and baggage layers – must bridge the requirements of instrumentation and tracing tools, by providing concrete data formats and propagation rules that are broad enough to preserve happened-before relationships between arbitrary events during execution. This provides a common underlying context representation on which I can implement execution-flow scoped variables, which encapsulated in BDL data types. These intermediary layers have three important requirements:

Expressive Representation At instrumentation time, opaque contexts shield developers from the details of tracing tools. Similarly, execution-flow scoped variables shield tracing tool developers from implementation and deployment

challenges. In particular, the representation must be expressive enough to correctly preserve the concurrent data types of different tracing tools, even when tracing tool logic is not present at a node. It must also be simple and efficient if we hope to reach consensus on its deployment across systems and tracing tools; it should serve as the "narrow waist" upon which different tracing tools can be built, analogous to the role of the IP layer in networking.

Multiplexed Tracing Tools The Tracing Plane needs to support multiple tracing tools side by side, due to the growing variety of tracing tools and use cases. Practitioners have described issues such as inadvertently clobbering other tools' key-value pairs, and how "high durability" use cases such as access control using data provenance need to be supported by end-to-end tracing frameworks, regardless of whether the framework decided to sample a trace of the execution [243].

Overflow When a context grows too large and exceeds the size constraints of a system, I call this *overflow*. For some tracing tools overflow is not a consideration because they only propagate fixed-size contexts [125, 162, 204]. However, overflow is important for dynamic contexts [110, 130]. For example, Pivot Tracing cannot prevent queries from propagating a large number of tuples (cf. Chapter 4); similarly, the graph of latency predictions that DQBarge propagates would be prohibitively large for, *e.g.*, the entire Facebook processing path [110]. Overflow touches on all layers of the Tracing Plane: size limits are specified at the instrumentation level; implemented in the intermediary layers; and must be exposed to tracing tools, *i.e.* so that they can later detect that overflow occurred. I use the term *trimming* to describe constraining context sizes, *e.g.* by dropping data.

5.5 Contributions

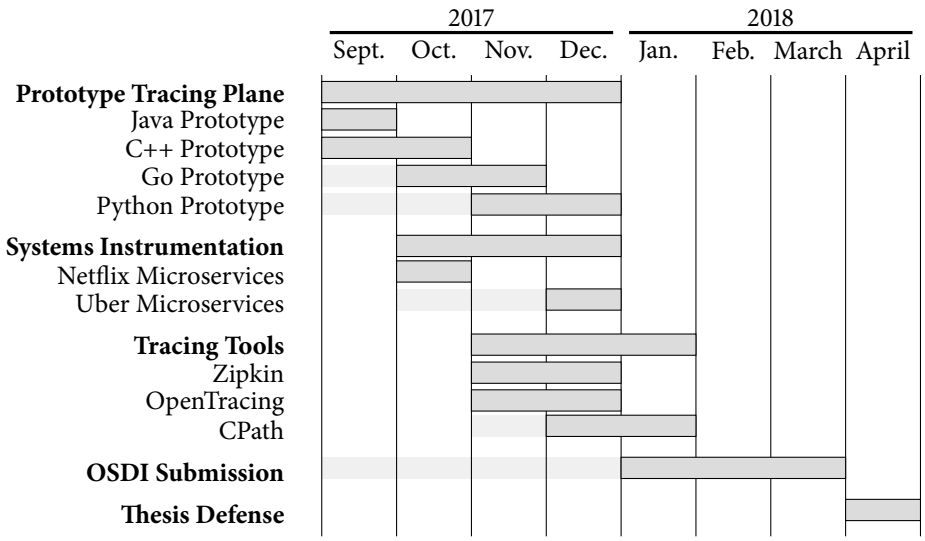
To demonstrate and evaluate the Tracing Plane, I will implement a prototype in Java, C++, and Go. I will implement revised and extended versions of X-Trace [126], Zipkin [233], OpenTracing [182], Retro (cf. Chapter 3), and Pivot Tracing (cf. Chapter 4) that are backed by the Tracing Plane, plus new tracing tools for resource management and critical path analysis. I will illustrate how the Tracing Plane's layered design makes it easy for different tracing tools to coexist simultaneously, and how deploying new tools requires no further instrumentation for propagating contexts.

The main contributions of this work will be follows:

- Execution-flow scoped variables, an abstraction that simplifies developing and deploying tracing tools;
- Baggage contexts, an abstraction for opaque, reusable, and generic instrumentation of distributed systems;
- An intermediary representation of contexts that encapsulates the subtleties of distributed tracing and enables coexistence of multiple tracing tools;
- Baggage Definition Language, and its compiler, for creating and using execution-flow scoped variables.

5.6 Timeline

The following table summarizes the timeline for the research deliverables of the proposed work.



Appendix A

Supplementary Tables

Company	*Services*	*Engineers*	*Tools*	*Use Cases*
Allegro	250+	500	Zipkin†*	debugging; understanding service dependencies; network traffic analysis; latency monitoring
BBN Technologies	30+	60	Zipkin†	understand service dependencies; performance and latency monitoring
Coursera	15+	60	Zipkin†*	dependency visualization; failure correlation and analysis
Etsy	—	200+	CrossStitch† [244]	latency monitoring; aggregated analysis
Facebook	—	—	FBTrace	mobile analysis; regression analysis
FINN.no	200	120	Zipkin†	
Google	—	—	Dapper [220], Census [130]	performance and resource monitoring; security auditing; root-cause analysis
Groupon	400+	1700	Zipkin†	performance improvements; architectural understanding; monitoring; SLA enforcement; anomaly detection; ad-hoc exploratory analysis
Hailo	200+	30	*In-House*† [135]	debugging; metric aggregation; architectural understanding; network traffic analysis; performance optimizations
Line	24+	200+	Brave, Zipkin†	latency monitoring; metrics monitoring
Lookout	15+	100	Zipkin†	statistics and metrics monitoring; deployment tooling; client whitelisting;
Lyft	—	—	zend† [152]	dependency analysis; latency analysis; mobile device correlations
Medidata Solutions	100	—	Zipkin†	system monitoring
Naver	100	2000	Naver Pinpoint†	architectural understanding; realtime monitoring; stacktrace sampling; batch analysis
Netflix	100+	1000+	Salp† [145]	dependency analysis; ad-hoc offline querying; realtime analysis; critical path analysis
Pinterest	—	—	PinTrace† [147]	latency analysis; architectural understanding; debugging; cost attribution; root-cause analysis
Prezi	50	100	Zipkin†	latency analysis; service dependency analysis
SmartThings	24+	35	Zipkin†	real-time analysis
SoundCloud	50	140	Zipkin†	architectural understanding, performance optimizations; latency analysis; batch analysis
Sourcegraph	—	—	Appdash†	debugging; performance and latency monitoring
Tracelytics	—	—	TraceView§	latency analysis; performance monitoring; realtime monitoring; metric aggregation
TomTom Maps	10+	100+	Brave, Zipkin†	statistical analysis and aggregation
Uber	2000+	2000+	Jaeger	architectural understanding, execution clustering; historical analysis; anomaly detection; inspect service dependencies; latency correlations; real-time aggregations
Yelp	300+	—	Zipkin† [2]	debugging; service dependency analysis; latency analysis
Zalando	100+	1000+	Zalando Tracer§	realtime and batch analysis
Zhihu	150+	80+	Zipkin†	architectural understanding; metric aggregation; dependency analysis; stack trace analysis; latency analysis

version with extensions or modifications †Dapper [220] derivative §X-Trace [126] derivative

Table A.1: Information about tracing tools deployed at several companies. Information was gathered from the Distributed Tracing Workgroup [117]

Bibliography

[1] Adrian Cockroft, Amazon Web Services. Personal Communication. (February 2017). Page 14.

[2] Prateek Agarwal. Distributed Tracing at Yelp. (April 2016). Retrieved January 2017 from https://engineeringblog.yelp.com/2016/04/distributed-tracing-at-yelp.html. Page 73.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Pages 9 and 10.

[4] Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013. Page 27.

[5] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval Tree Clocks: A Logical Clock for Dynamic Systems. In *12th International Conference On Principles Of Distributed Systems (OPODIS '08)*. Pages 13 and 58.

[6] Nuha Alshuqayran, Nour Ali, and Roger Evans. A Systematic Mapping Study in Microservice Architecture. In *9th IEEE International Conference on Service-Oriented Computing and Applications (SOCA '16)*. Pages 4, 5, 8, 9, and 12.

[7] Amazon. Amazon Web Services. Retrieved January 2017 from https://aws.amazon.com. [Online; accessed January 2017]. Page 4.

[8] Amazon. AWS Lambda. Retrieved January 2017 from https://aws.amazon.com/lambda. Page 14.

[9] Amazon. Summary of the Amazon DynamoDB Service Disruption. (September 2015). Retrieved June 2016 from https://aws.amazon.com/message/5467D2/. Page 5.

[10] Andrew Wang, Cloudera. Personal Communication. (March 2014). Page 21.

[11] Apache. Accumulo. Retrieved January 2017 from https://accumulo.apache.org/. Pages 14, 17, and 46.

[12] Apache. ACCUMULO-1197: Pass Accumulo trace functionality through the DFSClient. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-1197. Page 14.

[13] Apache. ACCUMULO-3507: NamingThreadFactory.newThread should not wrap runnable with TraceRunnable. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3507. Page 13.

[14] Apache. ACCUMULO-3725: Majc trace tacked onto minc trace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3725. Page 13.

[15] Apache. ACCUMULO-3741: Reduce incompatibilities with htrace 3.2.0-incubating. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3741. Page 15.

[16] Apache. ACCUMULO-4171: Update to htrace-core4. https://issues.apache.org/jira/browse/ACCUMULO-4171. [Online; accessed January 2017]. Page 15.

[17] Apache. ACCUMULO-4191: Tracing on client can sometimes lose "sendMutations" events. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-4191. Page 13.

[18] Apache. ACCUMULO-4192: Analyze Threading for Tracing correctness. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-4192. Page 13.

[19] Apache. ACCUMULO-898: Look into replacing CloudTrace. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-898. Pages 14 and 15.

[20] Apache. Accumulo CloudTrace. Retrieved January 2017 from http://accumulo.apache.org/1.6/accumulo_user_manual.html#_tracing. Page 14.

[21] Apache. Cassandra. Retrieved January 2017 from https://cassandra.apache.org/. Pages 13 and 14.

[22] Apache. CASSANDRA-10392: Allow Cassandra to trace to custom tracing implementations. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-10392. Pages 13 and 15.

[23] Apache. CASSANDRA-1123: Allow tracing query details. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-1123. Page 14.

[24] Apache. CASSANDRA-11706: Tracing payload not passed through newSession(..). Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-11706. Page 13.

[25] Apache. CASSANDRA-12835: Tracing payload not passed from QueryMessage to tracing session. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-12835. Pages 13 and 14.

[26] Apache. CASSANDRA-5483: Repair tracing. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-5483. Pages 13 and 15.

[27] Apache. CASSANDRA-7644: Tracing does not log commitlog/memtable ops when the coordinator is a replica. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-7644. Page 13.

[28] Apache. CASSANDRA-7657: Tracing doesn't finalize under load when it should. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-7657. Page 13.

[29] Apache. CASSANDRA-8032: User based request scheduler. Retrieved June 2016 from https://issues.apache.org/jira/browse/CASSANDRA-8032. Page 19.

[30] Apache. CASSANDRA-8553: Add a key-value payload for third party usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-8553. Page 15.

[31] Apache. CLOUDSTACK-618: API request throttling to avoid malicious attacks on MS per account through frequent API request. Retrieved June 2016 from https://issues.apache.org/jira/browse/CLOUDSTACK-618. Page 19.

[32] Apache. Cloudstack API Request Throttling. Retrieved June 2016 from https://cwiki.apache.org/confluence/display/CLOUDSTACK/API+Request+Throttling. Pages 5 and 18.

[33] Apache. HADOOP-13438: Optimize IPC server protobuf decoding. Retrieved January 2017 from https://issues.apache.org/jira/browse/HADOOP-13438. Page 13.

[34] Apache. HADOOP-13473: Tracing in IPC Server is broken. Retrieved January 2017 from https://issues.apache.org/jira/browse/HADOOP-13473. Page 13.

[35] Apache. HADOOP-3810: NameNode seems unstable on a cluster with little space left. Retrieved June 2016 from https://issues.apache.org/jira/browse/HADOOP-3810. Pages 5 and 18.

[36] Apache. HADOOP-6599 Split RPC metrics into summary and detailed metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6599. Pages 16 and 45.

[37] Apache. HADOOP-6859 Introduce additional statistics to FileSystem. Retrieved July 2017 from https://issues.apache.org/jira/browse/HADOOP-6859. Pages 16 and 45.

[38] Apache. HADOOP-9640: RPC Congestion Control with FairCallQueue. Retrieved June 2016 from https://issues.apache.org/jira/browse/HADOOP-9640. Page 19.

[39] Apache. HBase. Retrieved June 2016 from https://hbase.apache.org. Pages 14, 20, and 58.

[40] Apache. HBASE-11004: Extend traces through FSHLog#sync. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-11004. Page 13.

[41] Apache. HBASE-11559 Add dumping of DATA block usage to the BlockCache JSON report. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-11559. Pages 16 and 45.

[42] Apache. HBASE-11598: Add simple RPC throttling. Retrieved June 2016 from https://issues.apache.org/jira/browse/HBASE-11598. Page 19.

[43] Apache. HBASE-12356: Rpc with region replica does not propagate tracing spans. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-12356. Page 14.

[44] Apache. HBASE-12364 API for query metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12364. Pages 16 and 45.

[45] Apache. HBASE-12424 Finer grained logging and metrics for split transaction. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12424. Pages 16 and 45.

[46] Apache. HBASE-12477 Add a flush failed metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12477. Pages 16 and 45.

[47] Apache. HBASE-12494 Add metrics for blocked updates and delayed flushes. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12494. Pages 16 and 45.

[48] Apache. HBASE-12496 A blockedRequestsCount metric. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12496. Pages 16 and 45.

[49] Apache. HBASE-12574 Update replication metrics to not do so many map look ups. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-12574. Pages 16 and 45.

[50] Apache. HBASE-12938: Upgrade HTrace to a recent supportable incubating version. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-12938. Page 15.

[51] Apache. HBASE-13077: BoundedCompletionService doesn't pass trace info to server. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13077. Page 13.

[52] Apache. HBASE-13078: IntegrationTestSendTraceRequests is a noop. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13078. Page 14.

[53] Apache. HBASE-13458: Create/expand unit test to exercise htrace instrumentation. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-13458. Page 14.

[54] Apache. HBASE-14451: Move on to htrace-4.0.1 (from htrace-3.2.0) and tell a couple of good trace stories. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-14451. Pages 12 and 15.

[55] Apache. HBASE-15880: RpcClientImpl#tracedWriteRequest incorrectly closes HTrace span. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-14880. Page 13.

[56] Apache. HBASE-2257 [stargate] multiuser mode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-2257. Pages 16 and 45.

[57] Apache. HBASE-4038 Hot Region : Write Diagnosis. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4038. Pages 16 and 45.

[58] Apache. HBASE-4145 Provide metrics for hbase client. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4145. Pages 16, 17, and 45.

[59] Apache. HBASE-4219 Add Per-Column Family Metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-4219. Pages 16 and 45.

[60] Apache. HBASE-6215: Per-request profiling. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6215. Page 14.

[61] Apache. HBASE-6449: Dapper like tracing. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-6449. Pages 9, 12, and 14.

[62] Apache. HBASE-7958 Statistics per-column family per-region. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-7958. Pages 16 and 45.

[63] Apache. HBASE-8370 Report data block cache hit rates apart from aggregate cache hit rates. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8370. Pages 16 and 45.

[64] Apache. HBASE-8868 add metric to report client shortcircuit reads. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-8868. Pages 16 and 45.

[65] Apache. HBASE-9121: Update HTrace to 2.00 and add new example usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-9121. Page 15.

[66] Apache. HBASE-9722 need documentation to configure HBase to reduce metrics. Retrieved July 2017 from https://issues.apache.org/jira/browse/HBASE-9722. Pages 16 and 45.

[67] Apache. HBase Reference Guide. Retrieved July 2017 from https://hbase.apache.org/book.html. Pages 16 and 45.

[68] Apache. HDFS-10174: Add HTrace support to the Balancer. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-10174. Page 13.

[69] Apache. HDFS-11622 TraceId hardcoded to 0 in DataStreamer, correlation between multiple spans is lost. Retrieved April 2017 from https://issues.apache.org/jira/browse/HDFS-11622. Page 12.

[70] Apache. HDFS-4169 Add per-disk latency metrics to DataNode. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-4169. Pages 16 and 45.

[71] Apache. HDFS-4183: Throttle block recovery. Retrieved June 2016 from https://issues.apache.org/jira/browse/HDFS-4183. Pages 5, 18, and 22.

[72] Apache. HDFS-5253 Add requesting user's name to PathBasedCacheEntry. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-5253. Pages 16 and 45.

[73] Apache. HDFS-5274: Add Tracing to HDFS. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-5274. Pages 12, 13, and 14.

[74] Apache. HDFS-6093 Expose more caching information for debugging by users. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6093. Pages 16 and 45.

[75] Apache. HDFS-6268 Better sorting in NetworkTopology.pseudoSortByDistance when no local node is found. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6268. Pages 59, 60, and 62.

[76] Apache. HDFS-6292 Display HDFS per user and per group usage on webUI. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-6292. Pages 16 and 45.

[77] Apache. HDFS-7054: Make DFSOutputStream tracing more fine-grained. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7054. Page 12.

[78] Apache. HDFS-7189: Add trace spans for DFSClient metadata operations. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7189. Page 13.

[79] Apache. HDFS-7390 Provide JMX metrics per storage type. Retrieved July 2017 from https://issues.apache.org/jira/browse/HDFS-7390. Pages 16 and 45.

[80] Apache. HDFS-7963: Fix expected tracing spans in TestTracing along with HDFS-7054. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7963. Page 14.

[81] Apache. HDFS-9080: update htrace version to 4.0.1. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9080. Pages 12 and 15.

[82] Apache. HDFS-945: Make NameNode resilient to DoS attacks (malicious or otherwise). Retrieved June 2016 from https://issues.apache.org/jira/browse/HDFS-945. Pages 5 and 18.

[83] Apache. HDFS-9853: Ozone: Add container definitions. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9853. Page 12.

[84] Apache. HTrace. Retrieved January 2017 from http://htrace.incubator.apache.org/. Pages 9, 17, and 46.

[85] Apache. HTRACE-330: Add to Tracer, TRACE-level logging of push and pop of contexts to aid debugging "Can't close TraceScope..". Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-330. Page 13.

[86] Apache. HTRACE-5: Tracing never ends when using TraceRunnable in a thread pool. Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-5. Page 13.

[87] Apache. HTrace Developer Guide. Retrieved January 2017 from http://htrace.incubator.apache.org/developer_guide.html. Page 14.

[88] Apache. KUDU-1395: Scanner KeepAlive requests can get starved on an overloaded server. Retrieved June 2016 from https://issues.apache.org/jira/browse/KUDU-1395. Pages 18 and 19.

[89] Apache. MESOS-1949 All log messages from master, slave, executor, etc. should be collected on a per-task basis. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-1949. Pages 16, 17, and 45.

[90] Apache. MESOS-2157 Add /master/slaves and /master/frameworks/{framework}/tasks/{task} endpoints. Retrieved July 2017 from https://issues.apache.org/jira/browse/MESOS-2157. Pages 16 and 45.

[91] Apache. PHOENIX-177: Collect usage and performance metrics. Retrieved January 2017 from https://issues.apache.org/jira/browse/PHOENIX-177. Page 12.

[92] Apache. Phoenix 195: Zipkin. Retrieved January 2017 from https://github.com/apache/phoenix/pull/195. Page 15.

[93] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. MacroBase: Analytic Monitoring for the Internet of Things. *arXiv preprint arXiv:1603.00567*, 2016. Page 10.

[94] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*. Page 19.

[95] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*. Pages 9, 10, and 52.

[96] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online Modelling and Performance-Aware Systems. In *9th USENIX Workshop on Hot Topics in Operating Systems (HotOS '03)*. Pages 9 and 10.

[97] Jeff Barr. AWS X-Ray – See Inside of Your Distributed Application. (December 2016). Retrieved January 2017 from `https://aws.amazon.com/blogs/aws/aws-x-ray-see-inside-of-your-distributed-application/`. Page 14.

[98] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *36th ACM International Conference on Software Engineering (ICSE '14)*. Page 10.

[99] Peter Bodik. Overview of the Workshop of Managing Large-Scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques (SLAML'11). *SIGOPS Operating Systems Review*, 45(3):20–22, 2011. Pages 15 and 44.

[100] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Pages 4, 17, and 19.

[101] Bryan Cantrill. Hidden in Plain Sight. *ACM Queue*, 4(1):26–36, 2006. Pages 16 and 45.

[102] Bryan Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic Instrumentation of Production Systems. In *2004 USENIX Annual Technical Conference (ATC)*. Pages 16, 45, 55, and 65.

[103] Anupam Chanda, Alan L Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM European Conference on Computer Systems (EuroSys '07)*. Pages 8, 9, 10, 19, and 63.

[104] Anupam Chanda, Khaled Elmeleegy, Alan L Cox, and Willy Zwaenepoel. Causeway: Support for Controlling and Analyzing the Execution of Multi-tier Applications. In *6th ACM/IFIP/USENIX International Middleware Conference (Middleware '05)*. Pages 11, 19, 65, and 67.

[105] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. Pages 14 and 58.

[106] Mike Y Chen, Anthony Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. Path-Based Failure and Evolution Management. In *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. Pages 9 and 10.

[107] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*. Pages 9, 10, 11, and 14.

[108] Shigeru Chiba. Javassist: Java Bytecode Engineering Made Simple. *Java Developer's Journal*, 9(1), 2004. Page 55.

[109] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. Pages 9, 10, 13, and 14.

[110] Michael Chow, Kaushik Veeraraghavan, Michael Cafarella, and Jason Flinn. DQBarge: Improving Data-Quality Tradeoffs in Large-Scale Internet Services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Pages 8, 9, 69, and 70.

[111] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *1st ACM Symposium on Cloud Computing (SoCC '10)*. Page 64.

[112] Datastax. JAVA-794: Enable tracing accross multiple result pages. Retrieved January 2017 from `https://datastax-oss.atlassian.net/browse/JAVA-794`. Page 13.

[113] Datastax. JAVA-815: No tracing results when a RETRY happens. Retrieved January 2017 from `https://datastax-oss.atlassian.net/browse/JAVA-815`. Page 13.

[114] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*. Page 59.

[115] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. Pages 4 and 17.

[116] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium (Security '11)*. Page 8.

[117] Distributed Tracing Workgroup. Distributed Tracing Workgroup. Retrieved January 2017 from `https://goo.gl/xs96fn`. Page 73.

[118] Thanh Do, Haryadi S Gunawi, Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Case for Limping-Hardware Tolerant Clouds. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*. Page 24.

[119] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. Page 63.

[120] Dynatrace. Dynatrace Application Monitoring. Retrieved July 2017 from `http://www.dynatrace.com`. Page 9.

[121] William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Page 8.

[122] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. Pages 16, 45, 53, 55, and 65.

[123] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, 3(5):10–14, 2016. Pages 4, 5, and 8.

[124] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. Page 9.

[125] Rodrigo Fonseca, Michael J Freedman, and George Porter. Experiences with Tracing Causality in Networked Services. In *2010 USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. Pages 13 and 70.

[126] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. Pages 8, 9, 11, 13, 14, 16, 23, 52, 54, 70, and 73.

[127] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-Resource Fair Queueing for Packet Processing. In *2012 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Pages 17 and 30.

[128] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. Pages 30 and 31.

[129] Google. Cloud Platform. Retrieved January 2017 from `https://cloud.google.com`. Page 4.

[130] Google. gRPC/Census. Retrieved January 2017 from `https://goo.gl/iEqlqH`. Pages 11, 69, 70, and 73.

[131] Oliver Gould. Real World Microservices: When Services Stop Playing Well and Start Getting Real. (May 2016). Retrieved July 2017 from `https://blog.buoyant.io/2016/05/04/real-world-microservices-when-services-stop-playing-well-and-start-getting-real/`. Pages 5 and 8.

[132] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. Page 40.

[133] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS '13)*. Pages 4, 5, 8, 18, and 22.

[134] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G^2: A Graph Processing System for Diagnosing Distributed Systems. In *2011 USENIX Annual Technical Conference (ATC)*. Page 9.

[135] Matt Heath. A Journey into Microservices: Dealing with Complexity. (March 2015). Retrieved January 2017 from `http://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/`. Pages 4, 5, 8, and 73.

[136] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. Gremlin: Systematic Resilience Testing of Microservices. In *36th IEEE International Conference on Distributed Computing Systems (ICDCS '16)*. Page 9.

[137] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*. Page 18.

[138] @Honest_Update. Honest status page. (October 2015). Retrieved July 2017 from `https://twitter.com/honest_update/status/651897353889259520`. Page 8.

[139] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *26th IEEE International Conference on Data Engineering Workshops (ICDEW '10)*. Page 64.

[140] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (ATC)*. Pages 4 and 20.

[141] Yurong Jiang, Lenin Ravindranath, Suman Nath, and Ramesh Govindan. WebPerf: Evaluating "What-If" Scenarios for Cloud-hosted Web Applications. In *2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Pages 9 and 14.

[142] Theodore Johnson. Approximate Analysis of Reader/Writer Queues. *IEEE Transactions on Software Engineering*, 21(3):209–218, 1995. Page 29.

[143] Henry R Kang. *Computational Color Technology*. SPIE Press Bellingham, 2006. Page 29.

[144] Sung-Il Kang and Heung-Kyu Lee. Analysis and Solution of Non-Preemptive Policies for Scheduling Readers and Writers. *SIGOPS Operating Systems Review*, 32(3):30–50, 1998. Page 29.

[145] Nitesh Kant. Distributed Tracing at Netflix. (July 2015). Retrieved January 2017 from `https://speakerdeck.com/niteshkant/distributed-tracing-at-netflix`. Page 73.

[146] Partha Kanuparthy, Yuchen Dai, Sudhir Pathak, Sambit Samal, Theophilus Benson, Mojgan Ghasemi, and PPS Narayan. YTrace: End-to-end Performance Diagnosis in Large Cloud and Content Providers. *arXiv preprint arXiv:1602.03273*, 2016. Page 9.

[147] Suman Karumuri. PinTrace: Distributed Tracing at Pinterest. (August 2016). Retrieved July 2017 from `https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup`. Pages 13, 14, and 73.

[148] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matti Hiltunen, Rajeev Gandhi, and Priya Narasimhan. Draco: Statistical Diagnosis of Chronic Problems in Large Distributed Systems. In *42nd IEEE/IFIP Conference on Dependable Systems and Networks (DSN '12)*. Pages 10 and 16.

[149] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP '01)*. Pages 27, 55, and 59.

[150] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming (ECOOP '97)*. Page 43.

[151] Tom Killalea. The Hidden Dividends of Microservices. *Communications of the ACM*, 59(8):42–45, 2016. Pages 4, 5, and 8.

[152] Matt Klein. Lyft's Envoy: From Monolith to Service Mesh. (January 2017). Retrieved January 2017 from `https://www.microservices.com/talks/lyfts-envoy-monolith-service-mesh-matt-klein/`. Page 73.

[153] Steven Y Ko, Praveen Yalagandula, Indranil Gupta, Vanish Talwar, Dejan Milojicic, and Subu Iyer. Moara: Flexible and Scalable Group-Based Querying System. In *9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*. Page 16.

[154] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *7th Biennial Conference on Innovative Data Systems Research (CIDR '15)*. Page 18.

[155] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. Pages 6, 11, 41, and 47.

[156] Brian Laub, Chengwei Wang, Karsten Schwan, and Chad Huneycutt. Towards combining online & offline management for big data applications. In *11th International Conference on Autonomic Computing (ICAC '14)*. Pages 58 and 63.

[157] Chris Li. eBay Tech Blog: Quality of Service in Hadoop. (August 2014). Retrieved June 2016 from `http://www.ebaytechblog.com/2014/08/21/quality-of-service-in-hadoop/`. Pages 5 and 18.

[158] Chris Li. HDFS NameNode Denial of Service Resilience. Retrieved June 2016 from `https://issues.apache.org/jira/secure/attachment/12616864/NN-denial-of-service-updated-plan.pdf`. Page 18.

[159] Ding Li, James Mickens, Suman Nath, and Lenin Ravindranath. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *6th ACM Symposium on Cloud Computing (SoCC '15)*. Page 9.

[160] Lightstep. Lightstep. Retrieved January 2017 from `http://lightstep.com`. Page 14.

[161] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, and Andrew Wang. Kudu: Storage for Fast Analytics on Fast Data. Retrieved June 2016 from `http://getkudu.io/kudu.pdf`. Pages 5 and 18.

[162] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. Pages 9, 11, 13, 14, 63, and 70.

[163] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Towards General-Purpose Resource Management in Shared Cloud Services. In *10th USENIX Workshop on Hot Topics in System Dependability (HotDep '14)*. Pages 13 and 14.

[164] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. Pages 9, 11, 13, 14, and 69.

[165] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-Dar. Modeling the Parallel Execution of Black-Box Services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*. Pages 9 and 10.

[166] Matthew L Massie, Brent N Chun, and David E Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004. Pages 16 and 45.

[167] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *2006 ACM SIGMOD International Conference on Management of Data*. Page 42.

[168] Haibo Mi, Huaimin Wang, Zhenbang Chen, and Yangfan Zhou. Automatic Detecting Performance Bugs in Cloud Computing Systems via Learning Latency Specification Model. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE '14)*, pages 302–307. IEEE. Page 9.

[169] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael R Lyu, and Hua Cai. Toward Fine-Grained, Unsupervised, Scalable Performance Diagnosis for Production Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1245–1255, 2013. Page 9.

[170] Haibo Mi, Huaimin Wang, Yangfan Zhou, Michael Rung-Tsong Lyu, Hua Cai, and Gang Yin. An Online Service-Oriented Performance Profiling Tool for Cloud Computing Systems. *Frontiers of Computer Science*, 7(3):431–445, 2013. Page 9.

[171] Microsoft. Azure. Retrieved January 2017 from `https://azure.microsoft.com`. Page 4.

[172] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I Seltzer. Provenance for the Cloud. In *8th USENIX Conference on File and Storage Technologies (FAST '10)*. Page 8.

[173] Andrew C Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In *16th ACM Symposium on Operating Systems Principles (SOSP '97)*. Page 8.

[174] Karthik Nagaraj, Charles Edwin Killian, and Jennifer Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. Page 16.

[175] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*. Pages 18 and 19.

[176] Naver. Pinpoint. Retrieved January 2017 from `https://github.com/naver/pinpoint`. Page 9.

[177] Netflix. Netflix Open Source Software. Retrieved January 2017 from `http://netflix.github.io/`. Page 4.

[178] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015. Pages 4, 5, and 8.

[179] Node.js. Continuation-Local Storage. Retrieved March 2015 from https://github.com/othiym23/node-continuation-local-storage. Page 8.

[180] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and Challenges in Log Analysis. *Communications of the ACM*, 55(2):55–61, 2012. Pages 17, 46, and 65.

[181] Adam J. Oliner, Ashutosh V. Kulkarni, and Alex Aiken. Using Correlated Surprise to Infer Shared Influence. In *40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. Pages 10 and 16.

[182] OpenTracing. OpenTracing. Retrieved January 2017 from http://opentracing.io/. Pages 9 and 70.

[183] OpenTracing. OpenTracing 28: Non-RPC Spans and Mapping to Multiple Parents. Retrieved January 2017 from https://github.com/opentracing/opentracing.io/issues/28. Page 12.

[184] OpenTracing. Specification 5: Non-RPC Spans and Mapping to Multiple Parents. Retrieved February 2017 from https://github.com/opentracing/specification/issues/5. Pages 12 and 15.

[185] OpenZipkin. B3-Propagation. Retrieved January 2017 from https://github.com/openzipkin/b3-propagation. Page 11.

[186] OpenZipkin. OpenZipkin 48: Would a common http response id header be helpful? Retrieved January 2017 from https://github.com/openzipkin/openzipkin.github.io/issues/48. Page 12.

[187] OpenZipkin. Zipkin 1189: Representing an asynchronous span in Zipkin. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1189. Page 12.

[188] OpenZipkin. Zipkin 1243: Support async spans. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1243. Page 12.

[189] OpenZipkin. Zipkin 1244: Multiple parents aka Linked traces. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1244. Page 12.

[190] OpenZipkin. Zipkin 925: How to track async spans? Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/925. Page 12.

[191] OpenZipkin. Zipkin 939: Zipkin v2 span model. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/939. Page 12.

[192] Oracle. The Java HotSpot Performance Engine Architecture. Retrieved March 2015 from http://www.oracle.com/technetwork/java/whitepaper-135217.html. Page 64.

[193] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing Latency in Multi-Tier Black-Box Services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*. Pages 9 and 10.

[194] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. Page 10.

[195] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Page 18.

[196] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *1998 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. Page 28.

[197] Abhay K Parekh and Robert G Gallagher. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Multiple Node Case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994. Page 25.

[198] Insung Park and Ricky Buch. Event Tracing: Improve Debugging and Performance Tuning with ETW. (April 2007). Retrieved July 2017 from `http://download.microsoft.com/download/3/A/7/3A7FA450-1F33-41F7-9E6D-3AA95B5A6AEA/MSDNMagazineApril2007en-us.chm`. [Online; published April 2007; accessed July 2017]. Page 65.

[199] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, (3):240–247, 1983. Pages 13 and 58.

[200] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating System Problems using Dynamic Instrumentation. In *2005 Ottawa Linux Symposium*. Pages 16 and 45.

[201] Lindsey C. Puryear and Vidyadhar G. Kulkarni. Comparison of Stability and Queueing Times for Reader-Writer Queues. *Performance Evaluation*, 30(4):195–215, 1997. Page 29.

[202] Ariel Rabkin and Randy Howard Katz. How Hadoop Clusters Break. *IEEE Software*, 30(4):88–94, 2013. Pages 16, 17, 45, and 46.

[203] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Osborne/McGraw-Hill, Berkeley, CA, USA, 2nd edition, 2000. Page 52.

[204] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Pages 8, 9, 19, 63, and 70.

[205] John Reumann and Kang G Shin. Stateful Distributed Interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, 2004. Pages 11 and 65.

[206] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. Pages 9 and 23.

[207] Javi Roman. The Hadoop Ecosystem Table. Retrieved January 2017 from `https://hadoopecosystemtable.github.io/`. Pages 4 and 14.

[208] Raja R. Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R. Ganger. So, you want to trace your distributed system? Key design insights from years of practical experience. Technical Report CMU-PDL-14-102, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213-3890, April 2014. Page 17.

[209] Raja R Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H Sigelman, Rodrigo Fonseca, and Gregory R Ganger. Principled Workflow-Centric Tracing of Distributed Systems. In *7th ACM Symposium on Cloud Computing (SOCC '16)*. Pages 13 and 14.

[210] Raja R Sambasivan, Alice X Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. Pages 9 and 10.

[211] Peter Schuller. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. (April 2014). Retrieved June 2016 from `https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html`. Page 19.

[212] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '11)*. Pages 7, 11, and 69.

[213] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. Technical Report, Inria–Centre Paris-Rocquencourt; INRIA, 2011. Page 69.

[214] Kai Shen and Meng Zhu. Best-Effort Request Labeling and Scheduling on Multicore Servers. Technical Report, University of Rochester, 2016. Page 11.

[215] Yuri Shkuro. Evolving Distributed Tracing at Uber Engineering. (February 2017). Retrieved July 2017 from https://eng.uber.com/distributed-tracing/. Page 9.

[216] Madhavapeddi Shreedhar and George Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996. Page 40.

[217] David Shue, Michael J Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. Pages 17, 18, and 19.

[218] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. Pages 4, 17, 20, 22, and 58.

[219] Benjamin H Sigelman. Towards Turnkey Distributed Tracing. (June 2016). Retrieved January 2017 from https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736. Page 13.

[220] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report, Google, 2010. Pages 8, 9, 10, 11, 12, 13, 14, 16, 17, 23, 46, 52, 54, and 73.

[221] SolarWinds. Traceview. https://traceview.solarwinds.com/. [Online; accessed July 2017]. Page 9.

[222] Spring. Spring Cloud Sleuth. Retrieved January 2017 from http://cloud.spring.io/spring-cloud-sleuth/. Page 9.

[223] Spring Cloud. Sleuth 306: Malformed X-B3 headers cause 500 error. Retrieved January 2017 from https://github.com/spring-cloud/spring-cloud-sleuth/issues/306. Page 15.

[224] Spring Cloud. Sleuth 410: Trace ID problem when using Spring ThreadPoolTaskExecutor. Retrieved January 2017 from https://github.com/spring-cloud/spring-cloud-sleuth/issues/410. Page 13.

[225] Spring Cloud. Sleuth 424: Not seeing traceids in the http response headers. Retrieved January 2017 from https://github.com/spring-cloud/spring-cloud-sleuth/issues/424. Page 12.

[226] Spring Cloud. Sleuth 425: Make Sleuth more robust in accepting invalid span headers. Retrieved January 2017 from https://github.com/spring-cloud/spring-cloud-sleuth/issues/425. Page 15.

[227] Dimitrios Stiliadis and Anujan Varma. Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998. Page 25.

[228] Hongkai Sun. General Baggage Model for End-to-End Tracing and Its Application on Critical Path Analysis. M.Sc. Thesis, Brown University, 2016. Pages 8, 9, and 69.

[229] Kun Suo, Jia Rao, Luwei Cheng, and Francis Lau. Time Capsule: Tracing Packet Latency across Different Layers in Virtualized Systems. In *7th ACM Asia-Pacific Workshop on Systems (APSys '16)*. Page 9.

[230] The Go Blog. Go Concurrency Patterns: Context. (July 2014). Retrieved January 2017 from `https://blog.golang.org/context`. Page 8.

[231] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-defined Storage Architecture. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. Pages 17, 18, and 25.

[232] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Pages 9, 10, 23, and 54.

[233] Twitter. Zipkin. Retrieved July 2017 from `http://zipkin.io/`. Pages 9, 14, 17, 46, and 70.

[234] Robbert Van Renesse, Kenneth P Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003. Page 16.

[235] Kenton Varda. Protocol Buffers: Google's Data Interchange Format. (July 2008). Retrieved January 2017 from `https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html`. Pages 27 and 55.

[236] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. Pages 18, 20, and 59.

[237] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *3rd ACM Symposium on Cloud Computing (SoCC '12)*. Pages 17, 18, 19, and 34.

[238] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Ion Stoica, and Randy Katz. Sweet Storage SLOs with Frosting. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*. Page 17.

[239] Chengwei Wang, Infantdani Abel Rayan, Greg Eisenhauer, Karsten Schwan, Vanish Talwar, Matthew Wolf, and Chad Huneycutt. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *13th ACM/IFIP/USENIX International Middleware Conference (Middleware '12)*. Pages 10, 17, 58, and 63.

[240] Chengwel Wang, Soila P Kavulya, Jiaqi Tan, Liting Hu, Mahendra Kutare, Mike Kasick, Karsten Schwan, Priya Narasimhan, and Rajeev Gandhi. Performance Troubleshooting in Data Centers: An Annotated Bibliography. *ACM SIGOPS Operating Systems Review*, 47(3):50–62, 2013. Page 16.

[241] Mick Semb Wever. Replacing Cassandra's tracing with Zipkin. (December 2015). Retrieved July 2017 from `http://thelastpickle.com/blog/2015/12/07/using-zipkin-for-full-stack-tracing-including-cassandra.html`. Page 15.

[242] Thomas Willhalm, Roman Dementiev, and Patrick Fay. Intel Performance Counter Monitor – A better way to measure CPU utilization. (January 2017). Retrieved July 2017 from `http://intel.ly/1C23e67`. Page 28.

[243] Distributed Tracing Workgroup. Tracing Workshop. (February 2017). Retrieved February 2017 from `https://goo.gl/2WKjhR`. Pages 9, 12, and 70.

[244] Paul Wright. CrossStitch: What Etsy Learned Building a Distributed Tracing System. (September 2014). Retrieved January 2017 from `https://www.slideshare.net/PaulWright9/crossstitch-what-etsy-learned-building-a-distributed-tracing-system-for-surge-conference-2014`. Pages 9 and 73.

[245] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. Pages 10 and 16.

[246] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *21st USENIX Security Symposium (Security '12)*. Page 8.

[247] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 159–172. ACM. Pages 16 and 45.

[248] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving Software Diagnosability via Log Enhancement. In *16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. Pages 15, 16, and 45.

[249] Yuri Shkuro, Uber. Personal Communication. (February 2017). Page 12.

[250] Denis Zaytsev. Distributed Tracing – The Most Wanted And Missed Tool In The Micro-Service World. (April 2016). Retrieved July 2017 from `https://medium.com/@denis.zaytsev/distributed-tracing-the-most-wanted-and-missed-tool-in-the-micro-service-world-c2f3d7549c47`. Page 9.

[251] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. Page 10.

[252] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. Pages 10 and 16.

[253] Jingwen Zhou, Zhenbang Chen, Haibo Mi, and Ji Wang. MTracer: A Trace-Oriented Monitoring Framework for Medium-Scale Distributed Systems. In *8th IEEE International Symposium on Service Oriented System Engineering (SOSE '14)*. Page 9.