LOG DISCOVERY, LOG CUSTODY, AND THE WEB INSPIRED APPROACH FOR OPEN DISTRIBUTED SYSTEMS TROUBLESHOOTING

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Nathaniel Kremer-Herman

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana

April 2021

LOG DISCOVERY, LOG CUSTODY, AND THE WEB INSPIRED APPROACH FOR OPEN DISTRIBUTED SYSTEMS TROUBLESHOOTING

Abstract

by

Nathaniel Kremer-Herman

Troubleshooting distributed systems is difficult due to the inherent complexities of runtime environments in computing resources utilized by the system. This is exacerbated in *open* distributed systems where membership of resources within the system is transient, placement of computations to resources is not known before runtime, and resources can be shared across multiple administrative jurisdictions (i.e. multiple independent clusters, clouds, or grids). It is infeasible to expect a user to know about these complexities. To make troubleshooting open distributed systems approachable by a user, the debug output of each individual *component* of the system (i.e. individual processes and services) must be *discoverable* and made *queryable*. However, contemporary approaches cannot provide these capabilities. Instead, they are provided using a novel architecture called TLQ (Troubleshooting via Log Query) which facilitates log discovery and custody for the user and allows them to directly query their system's debug output. In addition, it links components together when possible, inspired by the architecture of the World Wide Web. Through the lens of TLQ, this work presents a data model for debug output, a comparison of multiple querying approaches, a distributed querying architecture for open distributed systems troubleshooting, and considerations for more effective debug log design.

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

- Leslie Lamport

CONTENTS

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

When my distributed system does something unexpected, where do I look to find out why? As the use of large-scale, complex distributed systems in research and business applications continues to expand, this question has become increasingly important. Beyond the need to fix or optimize a live (and potentially critical) system, the question unveils a greater problem in transparency and comprehension of what goes on in a distributed system. Which pieces of the system are likely culprits for unexpected behavior? How can I access those, if I can at all? How do I make sense of their debug output? Each of these questions need to be answered before we can resolve why the system acted oddly in the first place. When troubleshooting issues on a single machine, all the resources are available to answer these questions. The behavior and its cause are co-located, and this makes it much easier to resolve bugs and problems.

However, troubleshooting a distributed system can be much more complex. It is not guaranteed the cause and effect of an issue are on the same machine. There may be many *components* (services and processes representing individual parts of the larger system) which interoperate. A hiccup in one component on Machine A can be passed along as faulty input to a component on Machine B and so on, potentially cascading throughout the whole system. Looking at the debug logs on Machine B may not give enough context to figure out that the component on Machine A was the problem.

In addition, distributed systems have added inherent complexity which a single

workstation does not experience. Dealing with the ordering of events and synchronization across machines [69], differing degrees of fault tolerance and recovery mechanisms [71], and determining the state of the whole system at a given time [18] are all added worries which make troubleshooting such a difficult prospect. This is especially true at larger scales where we may not be able to figure out by hand which components did what and where. We use troubleshooters and debuggers to help alleviate the complexities of troubleshooting a distributed system. Each tool we use brings us closer to answering our initial question. These can take the form of live monitoring tools which track the state of all components of a system, log analyzers which attempt to determine what happened after the fact, code injection methods which try to trace and possibly correct issues at the component level when they occur, replay tools which track state changes in the system and allow a step-by-step rerun of what happened, and tools which provide similar experiences to low-level debuggers like `gdb` or text search and querying like with `grep` but at scale. These methods each shed more context as to what was happening in a distributed system when it misbehaved.

While these types of tools provide the context necessary to figure out why a system is misbehaving, to varying degrees of detail and success, they are not sufficient for troubleshooting *all* distributed systems. If we are troubleshooting an *open* distributed system, the job becomes much more difficult. A distributed system is considered to be open if it has the following qualities: transient membership and quantity of resources, on-demand placement of computation to resources, and independent resource domains are allowed or utilized.

Transient membership means resources can come and go in the system. Losing a resource does not break the system, and adding a new one allows it to be utilized. Further, the quantity of available resources provided to the system from each machine is allowed to vary over time. For example, a user could directly log on to a machine. This may preemptively kill any components on that machine, effectively taking its

resources away from the system. Or, perhaps this user is given *some* but not all resources on the machine, so fewer components may run on it for the duration of that user's session. Once they log out, the machine (or the resources reserved on it) are added back to the system's available pool.

On-demand placement of computation to resources is in contrast of planned placement. Both are often handled by a scheduler service. In planned placement, each component is meticulously scheduled to land on particular machines *before* the system begins to execute computations. In an open system, it is not possible to know *a priori* on which machines each component will land since the scheduler places work on any available machine which fits the requirements of the component. This must be done at runtime, on-demand. Again, machines may come and go, so it is also not possible to plan out where the scheduler should make its placements beforehand.

Independent resource domains refer to multiple administrative jurisdictions which may be utilized to patch together an open system. Different cluster, cloud, and grid providers may each provide *some* of the resources used by the system as a whole. Because these machines are maintained by different organizations, they may not be able to directly communicate with each other. Further, machines from one administrative domain may not know that machines from other domains exist. A user-facing component like a scheduler or workflow manager is typically provided the necessary credentials to communicate across domains, however the majority of system components cannot.

Each of these added complexities makes troubleshooting an open distributed system difficult. A user typically does not know where their components have landed, so they cannot directly look at their debug logs. Further, they may not have direct access to those logs if the machine is either out of the system (and thus unreachable) or if it is in a separate domain which the user cannot access. Typically, a user is only aware of a single component of the system, which is user-facing like a work-

flow management system, job submitter, or interactive dashboard. Everything else is obscured to varying degrees by nature of computations and data changing hands multiple times among machines the user does not know exist. The categories of tools mentioned before are insufficient when used alone because they are at the mercy of these same complexities. They are not given the full context of an open system, so they cannot be used as the primary driver for troubleshooting them.

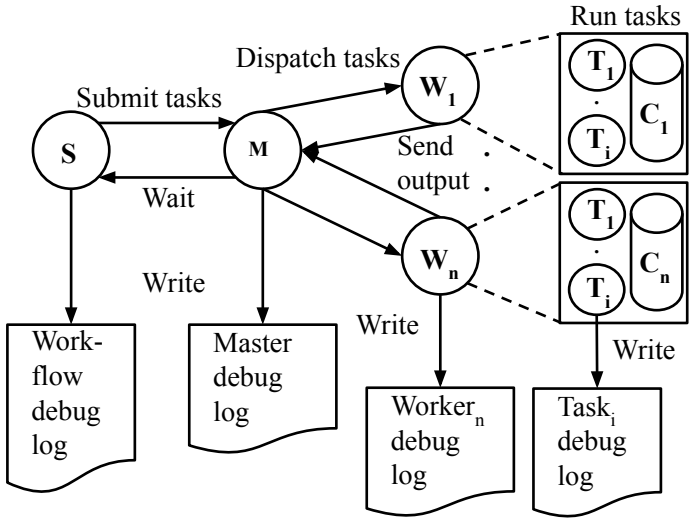

Figure 1.1. Scientific workflow architecture. The workflow manager (S) acts as a submitter and sends task definitions to the master process (M). The master dispatches tasks to workers (W). Workers run tasks (T) locally. Each component (S, M, W, T) writes to its own debug log file.

1.1 A Notional Example of Troubleshooting a Distributed System

To get a clearer insight as to why troubleshooting a distributed system can be difficult, we demonstrate a notional example. Consider a typical scientific workflow application which is being executed via a master-worker framework. The architecture of this distributed system is show in Figure 1.1. A workflow is composed of tasks which are definitions of work to be completed along with the necessary inputs and ex-

pected outputs. The workflow management system acts as a submitter process which sends tasks to the master process. This master then dispatches the task definition to a chosen worker process (which is typically on some other compute node) along with the input files specified. It is responsible for dispatching all tasks to available workers when possible. Assume a scale of $O(10,000)$ tasks. Each worker attempts to execute their current task's command, consuming any input files needed. This will in turn generate the expected output file(s) if successful. Specified inputs and outputs for each task are stored for reused at each worker's local cache. Since this is a scientific workflow, there most likely exist data dependencies between tasks. As such, tasks are only submitted by the workflow manager when their respective input dependencies are satisfied.

If a workflow fails because one task failed, how might we uncover this? We may use some monitoring dashboard, command line tools like `grep`, or choose to seek through debug logs manually. Regardless, we are most likely given only a piece of the puzzle for finding the true cause of failure. As we dissect the system, we will uncover more and more pieces in an iterative process of troubleshooting. This becomes intractable at larger scales (both the scale of the system and the number of failures).

The workflow management system is typically the only component the user can directly see. We can investigate the workflow management system log to see which task failed and that it was dispatched to the master process. The master process' log will show which worker ran that task. If we assume we can reach that worker's workspace sandbox (on another machine), we will see that its log will show the task that failed. We may even be told the task has its own log, or we may have to rerun the task locally and hope we can replicate the failure behavior.

This experience is clunky to the user since the onus is on them to know *which* logs are relevant to read and *where* each log is located. Monitoring dashboards and commonplace tools can help to reduce the overall time spent investigating these logs,

however they can only provide their functionality if the locations of each log are known. We assume the user can find each log in this notional example. However, in an open distributed system this is not the case. Since the user does not decide where their components execute (e.g. a batch job system handles this scheduling), nor are they typically told after the fact where these components were sent, an open distributed system adds an additional layer of complexity to this already tedious activity. It would be preferable to have a mechanism which can *query* the logs at their respective points of creation.

1.2 Effectively Troubleshooting Open Distributed Systems

This work describes an architecture for effectively troubleshooting open distributed systems called TLQ (Troubleshooting via Log Query). When using TLQ, a user is able to to access debug output which is transparently advertised to a user-facing client. Through TLQ, we provide mechanisms for log discovery (how to find debug logs), log custody (how to access those logs), and a rich querying experience for linked data (when one log references another) akin to the World Wide Web.

TLQ provides the key services of log discovery and log custody which make it possible to troubleshoot an open system. Servers at each machine used by the system advertise the existence and location of components and their debug output directly to the user, giving them a unique uniform resource locator (URL) for each log created by the system. These servers persist on each machine in the cluster, cloud, or grid, dedicated to monitoring components they are told about. In addition, they take custody of the logs created by components. If the server has permissions, it will take direct possession of the log, redirecting the default path in the component's command line string to a unique file in the server's working directory. If it does not, the server will periodically attempt to copy over updates to a local copy. This allows for logs to persist if the component which created it is ephemeral (incredibly short-lived).

Logs tracked by TLQ are periodically parsed, and the results are stored as JSON. These JSON documents can then be queried using the JX (JSON eXtended) language [114], providing a single interface for all logs. The querying possible with JX allows TLQ to provide a web approach to open systems troubleshooting. By this we mean JX queries can traverse links within one log to other logs, even if they are located on different machines. As parts of the query are resolved, its context may change multiple times depending upon how many links are fetched and evaluated, until a final list of JSON objects is returned as the fully evaluated result. This is accomplished by TLQ's assigning of URLs to every log advertised to log servers. In short, TLQ addresses the problems of log discovery and log custody while also providing an effective querying mechanism which allows us to answer our initial question: when something goes wrong in my system, how do I find out what happened?

1.3 Roadmap to TLQ

We have introduced why it is difficult yet so important to effectively troubleshoot distributed systems. To further compound the issue, we have shown why troubleshooting *open* distributed systems is an inherently more complex task. The scientific workflow notional example demonstrates how many different components in a system may need to be investigated to find the cause of a failure (though not all distributed system issues are failures). Further, it shows how tedious looking through logs by hand quickly becomes, especially as the scale of the system increases. Finally, we have provided a quick definition and outline of TLQ.

We will introduce a body of relevant related works spanning the different kinds of distributed systems problems which exist, tools (both contemporary and historic) which have been made to address these problems, and querying mechanisms which are relevant to TLQ's operation. From this foundation, we provide a specific, observed example of a common distributed system problem: proper resource scaling. We

provide a means of resolving this particular problem (a model called *capacity*) while also addressing how TLQ can provide more extensive support than the originally provided solution (an interactive web dashboard).

From there, we evaluate key pieces of TLQ's functionality from a fundamental level. We then present TLQ's architecture and two of its core pieces of functionality: log discovery and log custody. We define what these terms mean, how they help resolve the complexities of troubleshooting open distributed systems, and how TLQ guarantees the functionality of both. We then provide the final core functionality of TLQ: log querying. This is placed into the context of TLQ's web-inspired approach, showing how TLQ's JX queries are executed similarly to a request in the World Wide Web. Figure 1.2 portrays the layers of this work's contributions from most foundational (the bottom) to the culminating results (the top).

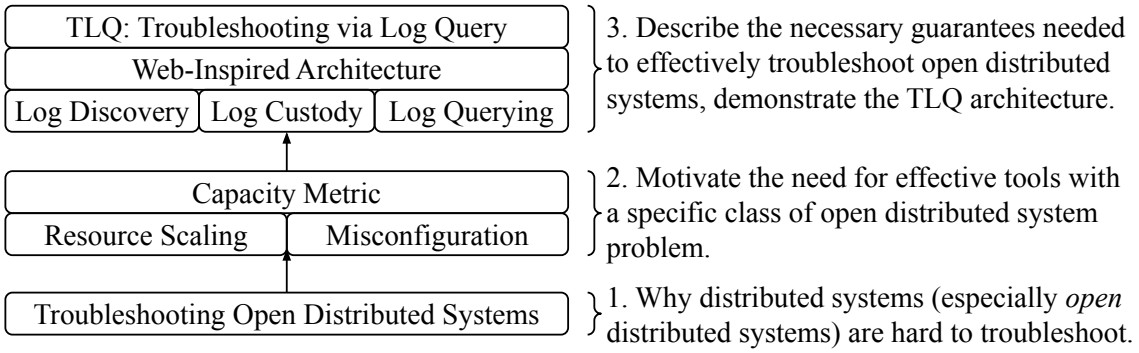

Figure 1.2. Concept map for TLQ.

CHAPTER 2

RELATED WORK

The complexity of distributed systems has long made troubleshooting them a difficult problem. By introducing parallel components and operations to a system, many underlying assumptions are altered. The concept of a single, unified state does not exist and cannot be reconciled. Concepts such as a unified time scale [69], taking a snapshot of the entire system state [18], and keeping a consistent, agreed-upon state [99] (as in the Byzantine Generals Problem [71] and the Part-Time Parliament Problem [70, 72]) are all complications inherent to distributed systems. These issues exist not only in clusters and grids but also in clouds [36] which can be even more complex since virtual machine instances in a cloud are generally treated as blackboxes to components communicating with them. Much work has been previously done to explore troubleshooting and debugging distributed applications in spite of these roadblocks which do not exist in traditional, serial debugging [95] noting at that time most users resorted to altering their components to explicitly print out state changes which becomes untenable even at small scales.

2.1 Early Attempts at Distributed Debugging

A comprehensive summary of many issues [15] outlines the aspects of distributed systems which previous, earlier works in distributed debugging have attempted to address. At a high level, these include: proper use of heterogeneous resources, concurrent execution both intra- and inter-machine, keeping sane distributed state information, and recovering from partial failures. Many of these issues (such as misconfig-

urations, fault tolerance, and poor concurrent performance) were also investigated in a longterm study of how to trace complex distributed systems [107] with an emphasis on doing performant tracing to reduce overhead on the system under study.

Recording the execution of a parallel application for later replay has historically been a method for troubleshooting nondeterministic behavior [28, 135] however the overhead for recording *complex* distributed applications can be significant. An initial work [74] investigated performance overheads for replay systems. Particularly, they demonstrate how polling a component may produce a complete enough history of events to meaningfully replay the system instead of exhaustively logging all events. EREBUS [32] provided an interesting twist on replay mechanisms. Instead of logging all nondeterministic events for an accurate recreation of the system under study, EREBUS executes two concurrent instances of the system. The first system performs as normal (including nondeterministic events which can be affected by monitoring by replay mechanisms). The second system runs in a deterministic fashion, waiting for the first system to produce the necessary state changes which are recorded and fed into the second system's workspace before proceeding in its execution. Another work [30] examines the then state-of-the-art approach of tracing a system's execution until an issue occurs, investigating the traces, then rerunning faulty components if needed to reproduce the issue. System replay is incredibly helpful since it stores a complete execution of the system under study. Any nondeterministic factors are captured and can be walked through deterministically within each replay. This is a capability other approaches cannot implement.

Earlier event-based handlers and online debugging tools attempted to provide a similar level of debugging to tools like `gdb`. One work [126] defined the differences between replay systems and event-based handlers. Whereas replay systems provide only observance of a system, event handlers institute control over the system. When a problematic event occurs, an event handler can stop the system to unpack at fine-

granularity what happened and why. EDL (Event Definition Language) [10, 11] was an early attempt at reducing the noise of potentially large logs. A user specifies types of events an underlying trace process records, allowing a user to filter and cluster only relevant component behaviors. This limits the number of breakpoints a user needs to worry about defining.

Other troubleshooters provided their own novel methods of making found problems transparent to the user. Node Prism [119] was an early attempt at specifying the scope of debug information that should be collected and operated upon. Node Prism allows a user to specify a set of resources (in this case CPU cores in a large-scale system) and remotely execute debugging commands upon monitor processes of those cores, providing a high level of interactivity at runtime. The TUMULT debugger [110] provided a means of creating a global, shared history of events in a distributed system. A key goal of this debugger was to keep all compiled programs as is rather than rely on users to alter their components. Comparing observed state changes to user defined global predicates [131] has historically been used to halt system execution when an mismatch occurs. This approach assumes the user knows the bounds their system's state should be within.

The scale and complexity of distributed computing today makes the use of these historical (and oftern lower-level) debuggers impractical since a user cannot make use of such low-level debugging output at contemporary scale during runtime anymore. This has been noted even in early parallel program tracers [56]. Communication delays make it difficult to produce a consistent, traced state of the system. Distributed systems are inherently nondeterministic at scale. By 1993, more than 200 bespoke and commercial parallel program debuggers existed [96] providing evidence that troubleshooting distributed systems has been a difficult problem for some time. These earlier works lay the foundation for more recent related works.

2.2 Misconfigurations and Testing

The heterogeneity of distributed systems noted in [15] necessitates a wider breadth of testing software and configuration management. Initially provided misconfigurations and failing to test for edge cases are both common issues for parallel applications. A body of related works [139–141] is focused on investigating the common causes of misconfiguration of a system. These can arise from poor documentation, incorrect assumptions by the developer about user behavior, and incomplete descriptions of the constraints a user can configure. One work [145] extols the benefits of providing simple testing in production database technologies and execution frameworks to find potentially critical bugs. Often times a user is not sure what options are available to them and how to appropriately use them [140]. Leaving configuration in users' hands without providing tools to help them detect misconfigurations places an undue burden on researchers, especially if they do not have the necessary technical knowledge to troubleshoot their issues manually [54].

Misconfigurations can lead to the breakdown of a complete cluster, like in Hadoop with misconfigurations being the most common cause of failure according to one study [104]. Besides developer bugs, no other category of failure came close to the number of misconfigurations encountered in the observed Hadoop cluster. Concerning troubleshooting Hadoop misconfigurations, one work [103] provides a way to precompute configuration errors in the system. By injecting faults into the system and analyzing runtime logs, failures due to improper customization and setup of the Hadoop cluster by system administrators are made clear. PCheck [141] was created to uncover misconfigurations which could cause damaging failures. PCheck analyzes source code and generates configuration values, seeing which instructions may be at fault for passing misconfigured values. EnCore [146] uses machine learning to find potential misconfigurations in a system. A training dataset of proper configurations are provided, and EnCore is then deployed in a production system. It will attempt to learn

the format of configurations it pulls from environment data available to it. Anomalies and mismatches are reported as outcomes of potential misconfigurations.

Misconfiguration failures compose a critical class of distributed system problems which TLQ seeks to address. We present a type of misconfiguration in Chapter 3 which spurred the development of TLQ. The TLQ architecture is designed to make the configurations of distributed components more transparent, assuming those details are logged. Environment variables and configuration files are two common ways to configure the execution environment of a component, both of which can be easily uncovered when using tracing programs on each component of the system under study. If configuration details are not logged by certain components, general purpose tools like TLQ would not be able to uncover these issues, leaving it up to domain-specific utilities.

2.3 Distributed System Analysis

One popular method of analyzing a distributed system is to inject code into parallel applications or server processes. This could be API calls to some tracing library for example. WAT (Why Across Time) provenance [136] injects code into a single component of a distributed system whereby a user can retrieve a minimal causal history (a hierarchical history of recorded events with explicit cause-effect relationships between events) for any given event meaning only the minimum number of operations required to replay that event would be returned. DySectAPI [52] provides users a means to modify their components to create probe trees (a graph capturing the causal history of logged events). The user modifies their application to make use of DySect's diagnostic functionality which is passed back to the user at runtime. IronFleet [44] utilizes injected code to verify a distributed system at a small scale such that critical bugs are uncovered *before* the system is scaled up. The purpose is to prove correctness for a system at a smaller yet non-trivial scale before implementing

the live system at its full scale.

Another suitable method of code injection is the inclusion of global state assertions and event-driven conditional statements. Meld [128] provides a method for users to specify expected outcomes of messages passed between components in a system. A user specifies certain message passing functions in their code as sources which are passed to Meld's online message verification process. Messages with faulty content or malformed headers are flagged as potential causes of issues in the system. Guard [2] allows for user-defined state assertions of system-created data throughout the system's execution. Failed assertions are collected and available for user inspection, providing context to which components may be faulty in a system. Panorama [48] provides an API which a user can use to modify individual components. When these modified components encounter a failure, it not only handles the failure as it would normally. It also reports to Panorama, providing a user with real-time notifications of observed failures. Rather than providing its own observers, Panorama relies on a user modifying their system such that each relevant component becomes its own observer of its state. However, these methods tend to favor applications and systems which can be tuned *during* execution. With many scientific applications this is unfavorable since alterations at runtime may affect results obtained. It is also possible the information the API calls are recording is already logged in the application [136].

Another related technique is not to inject code on a component but rather to inject faulty or noisy messages and events into a component at runtime. One work [100] presents a method of injecting faulty messages into a system as a stress test of components. The primary outcome of this method is the creation of a fault profile database which highlights which types of components are likely to experience faults given certain types of faulty messages which are processed by those components. NINJA [109] is a noise injection tool used to uncover message race conditions between communicating components. It was applied to MPI applications where communication within

a hierarchy of components is essential to ensure work is coordinated and completed properly. One work requires specifying breakpoints and instruction steps as flags for a debugger to log an event [85] which is also similar to direct code injection in its closeness to the source code of the system under study. However, one benefit of these flags is they allow the debugger to be largely system agnostic so long as the particular components being observed allow this level of interaction.

TLQ avoids this code injection approach in favor of monitoring components as they are. So long as they produce some form of debug log and tells TLQ about it, that component's log will be kept and tracked. This lowers the barrier to entry for users looking to add TLQ to their live system as no code needs to be modified. TLQ can also serve users who *cannot* modify the source code of their components. Code injection can also be used in tandem with TLQ since it will track the logs and enable querying of them while the code injection will add more details to the produced log.

Another way to analyze a distributed system is through runtime monitoring. This is particularly useful in large systems which run continuously. NetSight [42] monitors at the network level. The packet histories it provides can be helpful in troubleshooting faulty interactions between components. In this case components are network agents (such as a user client and a server) in communication with each other. Another packet analysis method [86] is aimed to assist cloud infrastructure providers with greater transparency regarding the structure of applications running on their resources. The individual processes (and even virtual machine instances) are linked together by the tracer when they communicate (whereas by default they are treated as blackboxes by the cloud infrastructure), providing a clear graph of all connected components in a cloud. SNooPy [149] is a system designed to provide users transparency of network provenance. It provides a user with the causal reasoning for why individual components are in a given state, providing clarity to the effects of interactions between components. ViSiDiA [88] is a framework for capturing snap-

shots of asynchronous networks, building off the distributed snapshot problem first discussed in [18]. In addition to taking snapshots of global state in the network, it also performs a comparison to user-defined predicates of what the state should be. This provides a level of correctness detection in the network useful for debugging faulty messages between components. NetCheck [150] collects traces of communication events between network hosts. It attempts to create a global ordering of events across the network (i.e. create a single state of a distributed system) and compares the events logged to an idealized model. Events that do not match the expected behavior according to the model are flagged as potential causes of issues encountered at runtime. Another network debugger, called ndb [41], provides backtracing functionality for packet histories. Its goal is to make more transparent the origins of faulty or redundant communications of components in a network.

Expanding from networking to distributed systems at large, a survey [108] of tracing distributed systems looks at the design decisions which must be made when designing tracing software. They note there are inherent attributes of distributed systems (such as component interaction which leaves the tracer's context) and how metadata is needed to link one component's trace to another to reveal meaningful relationships between them. We provide this need with TLQ although we do not encapsulate this linking process into a single tracing debugger. STAT [6] analyzes stack traces of highly concurrent applications, emphasizing performance given large scale input data. For the user, it provides a scalable visual presentation of call graphs which can be explored, reducing the chance of information overload from potentially thousands of individual traces by collapsing views of traces into more manageable, abstract representations until the user wants to investigate in more detail. One work investigates runtime tracing [120] and correlating related traces together (when components interact). One problem they encounter, which we also experienced with TLQ, is how to overcome parsing wildly different log formats to produce a consistent

view of a system. Pivot Tracing [83] allows users to specify at runtime certain metrics in their system. Messages between components are intercepted, and a causal history is built for each component's state changes. Messages which match the defined metrics of the user are presented to them. They can then filter and group the events contained in these messages for runtime inspection of their system. Another work [23] compares a trace of an active component to a reference trace of that component. The reference is used to determine if unexpected events occur in the active component. This was applied specifically to the case of a user moving their system from one platform to another, where environmental differences can greatly affect outcomes of a system.

One work [9] extends a traditional debugger (Ladebug) to parallel applications. This debugger works at the system level, providing the user with stack traces of the components in their system. An emphasis is placed on aggregating similar messages in the traces so as not to overwhelm the user. One work presented a utility which merges stack traces [75] in order to identify performance bottlenecks and sub-optimal configurations of a petascale system. BorderPatrol [61] is a system which collects traces of components, creating links between components which interact. Each traced process is considered a blackbox, and BorderPatrol's only interactions with it is to consult its internally produced trace log. Dapper [117] is a tracing infrastructure to realize causal and temporal traces of system events. The goal is to make clear the links between components at Internet scale, focusing primarily on network communications between components across machines but allowing a variety of events to be logged. Fay [26] is another platform for collecting component execution traces. Fay collects traces across a system by way of probe modules. These modules report traces to a centralized, user queryable aggregation of log data. PGDB [25] provides a low-level, `gdb`-like experience for MPI applications. Data from distributed backtraces are sent to a front-end machine to provide user interaction which is conceptually similar to TLQ's querying mechanism.

TLQ makes extensive use of system level component traces though it is not itself a tracing system. Nor is its primary function to gather traces at a central location. We present in Chapter 5 how tracing components' executions is beneficial to troubleshooting distributed systems and later apply this to TLQ specifically in Chapters 6 and 8.

While runtime tracing and reporting is beneficial for presenting the state of the system at any given time, other tools provide more interactive debugging or provide higher-level alerts of faults in the system under study. Falcon [78] creates a network of *spies* (watchdog processes monitoring components). When a spy process detects a failure, Falcon is made aware. If needed, Falcon has permissions to kill faulty components for the preservation of the system as a whole. Dustminer [60] provides troubleshooting for bugs in a network of sensor devices. Particularly, it uncovers repeated series of events which are accompanied by faulty behavior, allowing a user to see the series of events which may be producing a bug. Ganesha [93] attempts to provide logging of communication endpoints between blackboxed components in MapReduce systems. Machine learning was applied to training data to instruct Ganesha of ideally operating Hadoop applications. Components in a live system which do not conform to the conditions of the training data are flagged to the user as potential causes of failure without having to alter the components' code (maintaining they are blackboxes). One approach [134] provides real-time log parsing to discover links between components in a system. These links and the logs containing them are inserted into a graph representation of the system. These logs can then be more effectively mined for debug information, similar to how TLQ provides log discovery and log custody (presented in Chapter 6). Tools which can make use of the links between logs can take advantage of the graph structure to investigate relevant, related components when looking for the cause of failures.

SAMC (Semantic-Aware Model Checking) [77] is a principle for discovery of bugs

in cloud systems. Messages between components are intercepted by SAMC. Generic patterns for message types are used to determine if a message contains faulty state information. If so, it is flagged as a potential cause of failure. SAMC tracks how many times this flagged message occurs in the system, producing for the user a summary of the most common faulty messages rather than providing *all* faulty messages (avoiding information overload). DEMi [112] is a tool for minimizing the amount of log data presented to a user when troubleshooting a faulty execution of a system. When a fault occurs, the components involved are re-executed (if possible). The original faulty execution is compared to the re-execution. If the fault occurs again, that component is provided as a potential cause of the fault rather than providing the user with a log of *all* component events. One work [27] provides a comparison check to find data dependence faults in distributed applications. A serial implementation of the application is run alongside the distributed system under study as a reference where all data dependencies are satisfied since all data exists in the same locality. When data races or atomicity violations occur during execution, the user is notified.

SEI (Scalable Error Isolation) [13] is an algorithm for detecting arbitrary state corruption. State corruption can result in disastrous consequences as faulty state does not necessarily lead to outright crashes of components. Instead, faulty state can propagate across a system, doing more damage. Beyond detection, SEI also prevents data corruption from propagating through a system. One work [3] seeks to provide monitoring capabilities for OpenStack cloud systems. This monitor observes logs of multiple critical components in the system and alerts a user when a fault occurs. The emphasis is placed on components which would halt the system if a failure occurs as opposed to *all* components. D^3S [81] takes user provided predicates for the system under study. At runtime, component events which do not conform to these predicates are flagged. These events, called violations, are collected at a centralized log along with the state which was altered by these events. BigSift [34, 35] is designed to

create data provenance graphs for big data applications. Its goal is to identify faulty outputs created by components in a live system and provide the provenance history for them, making these faulty outputs transparent to a user at runtime rather than after the fact.

Lowgo [80] records dependencies between functions executed in cloud services. Users define calls to Lowgo's reporting mechanism in their serverless functions. These calls log when a function is executed. The log of these calls is used to create a causal history among the components of the system (including serverless function executions) which can then be investigated by the user. Antfarm [55] assists virtual machine monitors with system level events across virtual machines. A specific type of error, a mismatch between the virtual machine monitor's logging and an event occurring in the operating system, is made clear by Antfarm which provides transparency about processes in VMs. This allows the virtual machine monitor to more accurately log operating system events. CloudRanger [133] performs root cause analysis of cloud systems. It first analyzes logs for anomalies then creates a causal history of what led to that anomaly, allowing a user to identify with a high degree of certainty the root cause of an issue.

With TLQ, runtime analysis tools like those presented can be applied to open distributed systems whereas before many could not. Many works constrain their scope to well-defined system architectures or specific live systems. It should be noted that TLQ does not perform any application-specific log analysis as that would add undue complexity to its architecture (and other domain-specific tools may more readily provide that functionality when used with TLQ). Rather, when relevant, traditional and distributed debugging tools can be run alongside TLQ's architecture, with TLQ providing the locations of components allowing the runtime tools to perform their jobs in an open system.

A final method for analyzing a distributed system is after-the-fact log analysis.

Many log analyzers are geared toward monitoring resource usage by an application or for an entire system. Causal history analysis, which explores the history of component interactions and how unique sequences of interactions trigger specific events in a system, is a well understood technique for extracting potential sequences of events which may be the culprit(s) of failures in distributed systems and can take the form of event replay after-the-fact.

BITE [111] is a debugger which collects a history of component events for after-the-fact replay. It allows for replaying instructions step-by-step, multi-stepping, and fully replaying an execution of the system. Users are able to investigate the state of the system before and after the replay stage has been performed. Newt [82] is a framework for capturing data lineage between components of Hadoop and Hyracks systems. Users can query the resulting lineage graphs to pull out a subset of components which most likely caused faults, particularly those which created bad output due to using faulty inputs. Actoverse [115] is a debugger for distributed applications using the actor model (in which each component handles asynchronous message passing and no state is explicitly shared between components). It captures events for replay, providing a visual timeline of messages passed between components. The Intel Message Checker [24] is designed to debug MPI applications. It collects traces from components, performs an analysis, and highlights issues with deadlocks, performance bottlenecks, and resource allocation issues through a user visualization. Nemo [90] is a debugger which takes a more hands-on approach. When possible, it not only identifies issues but provides solutions to the user. It does this by identifying root causes of issues through a causal history, drawing the user's attention to faulty components in the system. SherLog [144] analyzes source code, taking debug logs as input, to identify areas of code which most likely triggered a failure in a system. It attempts to create links between logged events, creating a causal history of the system. Recon [76] is a framework for providing system replay and fine-grain querying. Once a

system has executed, a user can replay the events of each component with fine-grain instrumentation which provides an SQL-like querying mechanism for investigating relationships between components.

Concerning networks in particular, it can help to explore the travel history of packets and messages similar to a causal history of events in a parallel application. One work focuses on making failures in routing packets transparent to the source host [5]. These *packet obituaries* make delivery failures transparent to a user where they would normally be opaque, hidden potentially many hops outside the source host's jurisdiction. One work investigates routers as a treasure trove of debug information [102]. The focus of this work is providing a uniform representation of disparate router log formats (which vary between vendors), much like TLQ provides a uniform representation of log data at the component level. One network troubleshooting work presents how to find *missing* events using negative provenance to determine why a packet might not have reached its destination [138]. This concept of negative provenance is useful when comparing the observed events of a system with the expected behavior, showing when anticipated interactions do not occur.

Causal history is similar in concept to TLQ's approach to linking related components. Whereas a causal history identifies hierarchical relationships between components (such as parent-child), TLQ's approach creates links of components which interacted but makes no assumptions about the relationship between those components. This is introduced in detail in Chapter 6.

It is also common to apply some statistical model to the log data after-the-fact in order to extract anomalous events. Supercomputers are known to generate an abundance of system level log data, but extracting useful information from them can be an intractable problem given a large enough scale. One work [40] attempts to provide a more coherent view of these extremely dense log data by categorizing types of events using a Markov random field approach. These categories make the log

output more digestible to the user as compared to investigating the log by hand. This is a goal shared by TLQ. One work examines the difficulty of providing large-scale anomaly detection on system logs of many formats [53]. Their focus was on security violation detection in distributed systems, alerting the user of anomalous events in an interactive dashboard.

LOGAN [123] uses reference models of a system when analyzing system level logs of cloud applications. The user is given a report of normal, expected events and a list of events which resulted in either failure or were otherwise anomalous. Pip [106] reconciles differences between observed behavior of a system and its expected behavior (which is provided by the user). Behaviors which do not conform to expectations are flagged as potential causes of issues. The goal of Pip is to expose unexpected events, which may not necessarily be bugs or failures. PAL [89] is an anomaly localization system for cloud applications. PAL searches for anomaly propagation patterns, tracking down root causes of issues for these cloud applications. Another work focusing on anomaly detection [45] investigated the benefits of applying machine learning to log analysis in order to identify anomalous events. Their methods were applied to system level logs millions of lines long, pulling thousands of anomalous events when parsing through them. CORRMEXT [20] is a framework which analyzes resource usage logs and message logs to troubleshoot issues in clusters. A correlator tool attempts to establish links between resource usage and messages logged, attempting to determine a causal history for diagnosis of problems encountered. Statistical analysis has been applied to performance bottleneck diagnosis at the system call level [58] in large-scale cloud systems. One of its primary goals, like TLQ, is to increase transparency in systems under its observation.

Like the runtime monitoring systems presented, after-the-fact analysis can be applied to open systems when ran with TLQ. Again, we do not provide specific analysis of logs like these works. TLQ is more akin to a lookup system (with robust

querying capabilities of its own) which additional tools can be plugged into in order to provide their own, potentially domain-specific, functionality.

2.4 Distributed Debugging Visualizations

Often, visualizations of distributed systems are designed to show significant results or performance metrics of a parallel application. These visualizations are needed because often raw debug output is not easy to conceptualize [94]. Visualizations make the results, performance metrics, and debug output more digestible. One such visualization tool [33] makes the utilization of resources more transparent to the user. It establishes *moments* of utilization spread across the logged runtime of an application to provide a rough timeline of utilization. The Pegasus workflow management system [22] has built-in tools and a web-based performance dashboard. This dashboard is also able to pull up specific debug logs for the user on demand for their investigation. Another work demonstrates a compression of a parallel application's execution trace to show macro-scale behavioral motifs [84]. These motifs are collapsed representations of larger behaviors of an application which can be conceptualized as an identifiable segment of the whole workload. ParaGraph [46] visualizes a multitude of these performance metrics. One of ParaGraph's goals is to provide these graphs at different granularities (communications, tasks, and processors) which other works do not provide. The focus is generally not on troubleshooting these applications, rather it is to showcase what these applications do and how well they do it.

ENaVis [79] visualizes network activities produced by specific users and applications. This can result in a large-scale connectivity graph of relationships between activities which must be effectively managed. These relationships are based on logged communication which is similar to how TLQ uses messages between components kept in debug logs to establish links between components. Graphs showing performance metrics such as network connections made are provided for user investigation. Mini-

NAM [59] is another visualization for network activities to aid system administrators, this time focusing on packet flows. A relevant goal of MiniNAM is making load balancing issues clear to a user.

By going back further in the literature, one can find a collection of foundational works on visual distributed system debuggers. Xab [12] is one such tool which records events and provides a visual interface showing a monitoring dashboard. Like TLQ, Xab was designed to also interoperate with commonly available debugging tools. In this case, Xab is able to convert its records to formats used by other visualization suites to display performance and resource usage metrics. The MAD environment [62, 64] is designed to perform replay of recorded events like those described previously yet adds visualizations on top. MAD informs a user about their system's performance as well as displaying a graph of recorded events. ATEMPT [63] focuses solely on visualizing event graphs of recorded events in parallel applications. Specifically, it highlights detected errors from the recorded events.

While the visualization tools presented thus far are helpful for understanding an application's behavior (which is a great first step in troubleshooting), the level of interaction involved is not adequate for answering the question of why an application returned one result when another was expected. Visual *debuggers* range in granularity from system event traces to graphs showing process relationships and task relationships, and they can be used to help localize faults in production systems.

One work [21] provides a visualization for localizing faults in automotive distributed embedded systems. These small-scale computers help operate a vehicle system in tandem and are locally networked. When an issue occurs, providing a visualization of exactly where in the vehicle an embedded system has failed is an incredibly helpful form of troubleshooting. In the domain of web applications [16], it is helpful to visualize HTTP requests and failures due to HTTP accesses. Statistical analysis was included alongside visualization to perform anomaly detection of large-

scale web request logs. A more relevant work displays a graph representation of a master-worker framework [97] which displays the shared data across the master and workers and the unique data at each worker. Multiple visualizations are provided to make the structure of tasks executed by the framework and the structure of the framework itself more clear to a user.

Perhaps most relevant to TLQ are visualizations which help increase the transparency of a system's structure. This includes visualizing causal relationships and logged communications between components. Witt [19] is designed to visualize the execution of serverless applications. It graphs the execution timeline as presented by messages logged during runtime which provides context for the user about the behavior of their applicaiton. Mochi [124] is a visual log analysis tool for debugging Hadoop applications. It graphs the execution paths of each stage of MapReduce applications on Hadoop. These can be represented as specific tasks ran or as the behaviors executed (e.g. `MapTime`, `MapWriteTime`, or `ReduceTime`). SALSA [125] is another visual debugger made for MapReduce applications which focuses on causal traces of runtime events. Causal links are created between jobs which are then graphed in addition to performance metrics of the application. A survey of visualization techniques as applied to system performance [49] covers which visualizations are most descriptive of particular system performance metrics such as memory utilization and call graphs.

A general aim of these visualization tools is to increase transparency to the user (a goal of TLQ) and to highlight points of interest in the system which may otherwise be lost in the noise of the system's outputs, especially at large scale. One drawback of using TLQ by itself is that there is no interactive visualization of its system under study. The benefits of visualization tools and dashboards are well known (and advocated in these works).

2.5 Databases and Querying Techniques

TLQ uses distributed querying to make it possible to keep debug logs in place across an open distributed system. Database management systems and their respective querying techniques are incredibly relevant to how TLQ is designed. Indeed, choosing a querying language is a critical design decision [57] as it will be the primary means by which a user interacts with the querying engine and management technology. It must be comprehensible by the user, descriptive enough to write useful queries, and must present results in a format which is conducive to the user's exploration of data. A framework has been established [51] for choosing which query language best fits a use case. This framework names a few significant considerations such as the importance of the language's grammar, how queries are formulated, its usability, and its interaction model with a user.

The first iterations of TLQ were implemented using traditional database management technologies to varying degrees of success. Both SQL and NoSQL management technologies were used. Their performance and approach relative to each other have been extensively compared and are well understood [39, 98]. The high flexibility of NoSQL technologies is noted by [101] however traditional relational database technologies they investigated offered faster overall performance.

SQL variants have been demonstrated to be effective at large scale. In the case of Spanner [7], a NoSQL, schema-free management system was replaced at Internet-scale in exchange for the raw performance afforded by SQL. SQLGraph [122] is a technology which marries the relational database approach and graph traversal algorithms. They effectively store large-scale data in graph form while efficiently querying it with SQL. We show SQL applied to TLQ in Chapter 7.

The performance benefits of caching data at scale and tracking data provenance are well understood. Couchbase [17] is a database technology which focuses on effective caching of distributed, big data records stored in memory. P2 [118] is a diagnos-

tic system which performs tracing on a distributed system with built-in, performant querying of a cached database of logged events. While important, such performance considerations are not directly addressed in this work as they are outside the scope of TLQ's direct application.

Provenance debugging (querying the lineage of which processes led to data being created) is an effective method of finding root cause(s) of events in a distributed system. TLQ enables provenance troubleshooting inasmuch as the underlying debug logs created can be used to inform a user about the causal history of events. The SPADE project [31] provides a querying interface for decentralized data, like TLQ. Their focus is exclusively on provenance information. ExSPAN [148] also provides querying of provenance data however at larger, Internet-scale. In order to optimize their work, the authors investigated caching policies to reduce query roundtrip time. ProvBase [1] is a provenance storage and querying system designed specifically for scientific workflows which places provenance data at storage nodes on a network which clients can query.

Property graphs, a representation of system components tracking their properties (key-value pairs) and links to other components, have been the basis for multiple querying engines. PGQL [130] applies SQL-like syntax to property graph traversal like [122] however they focus on finding paths and determining reachability of records. Quegel [147] is designed as a property graph traversal and reachability querying engine focused on performance issues of traversing large graphs distributed across multiple machines. GoDB [50] thoroughly investigates common performance bottlenecks of property graph traversal engines such as pre-fetching data, queries performing in a distributed fashion, query path costs, and network communication costs. A similar approach creates links between hardware component interactions [91], making as few assumptions as possible about log structures, which can then be queried. In TLQ, we focus on software components and their interactions rather than hardware.

GraphQL [43] is a ubiquitously used property graph traversal engine which allows the developer to design their own query resolution mechanics which operate on top of the underlying GraphQL system. This flexibility was utilized in another early iteration of TLQ. The property graph approach is similar to TLQ's data model (presented in Chapters 6 and 8) which keeps track of links between components when interactions are logged and stores key-value pairs of metadata about each component.

In early prototypes of TLQ, we explored using SQL, NoSQL, and a graph traversal engine (directly applied to property graphs of debug information) as querying engines. However, we found the provided functionality did not meet the needs of TLQ nor did they apply cleanly to open distributed systems. As a consequence, we expanded the JX (JSON eXtended) [114] language to drive TLQ's querying capabilities. Its straightforward and easily modifiable implementation allowed for fine-tuning a query language which matched the open system use case. We discuss our choice of JX as the query language for TLQ in Chapter 7, which is compared to SQLite [92], RethinkDB [132], and GraphQL [43] as alternative implementations.

2.6 Semantic Web

The Semantic Web [14] is an expansion of the World Wide Web formally proposed in 2001. The goal is to encode metadata necessary to allow machines to reason over web data, uncovering the semantics of given data by exploring its relationships to metadata documents and other data. This includes expressive ways of storing data on the web, efficient querying mechanisms, and creating ontology networks [113]. These ontology networks provide taxonomies and classifications of data similar to class hierarchical structures in many programming languages.

Multiple languages have evolved to express semantic information about web content [4]. These languages primarily have three core mechanisms: provide descriptions of the concepts of a domain (classes), describe the relationships between concepts

(which may be hierarchical), and provide constraints on what can be expressed. The Resource Description Framework (RDF) [87] is a metadata model which enables web ontology. RDF data is represented by uniform resource identifiers (URIs). Expressions are made about data in the form subject-predicate-object where the subject is the data (called a resource in RDF), the predicate is the attribute of that resource being described, and the object is the value associated with that predicate. Multiple subject-predicate-object expressions can be made about any given resource. OWL [47] is a family of languages designed to represent ontologies. OWL provides the structure for how knowledge can be represented. In the Semantic Web, this means providing structure for resources and their relationships. OWL is designed to extend RDF's vocabulary, providing greater structure and descriptiveness of semantics associated with resources.

Effective querying methods built on top of these languages and semantic representations provide functionality to the data model. RDF establishes a graph representation of resources (specified by URIs) which are queryable. SPARQL [8, 29] is a Semantic Web query language for RDF. Its syntax is purposefully similar to SQL in that explicit relationships between resources enable rich queries to traverse the graph of resources. Starting with an initial class of resource (such as novels), a SPARQL query can traverse relationships (such as genre, author, or publisher) to find other resources within the same class (e.g. another novel by the same author) or to traverse different classes (e.g. the author was also an inventor and published multiple patents).

The goal of formalizing meaningful relationships between data is shared by TLQ though applied to debug logs rather than web documents. Further, the source of these debug logs may not have the necessary information to encode these meaningful relationships (or may choose not to include it) in debug output. TLQ makes a best effort attempt to find links between components from the information given to

it whereas the Semantic Web and web ontology frameworks enforce a schema and hierarchical structure more strictly, guaranteeing the necessary metadata is encoded in web data.

CHAPTER 3

TROUBLESHOOTING DISTRIBUTED SYSTEM PERFORMANCE USING SYSTEM CAPACITY AS A METRIC

As stated in Chapter 1, not all issues in a distributed system result in outright failure. In the case of performance troubleshooting, a system may be running slowly without an explicit failure occurring. This is an issue of resource provisioning (providing a system with an appropriate number of machines, CPU cores, memory, storage, etc.). When provisioning resources, a user often defines a certain number of resources to provide their system. Underlying mechanisms match the requested resources to the system when possible. However, a user may not know the best scale to run their system, and the behavior of their system may not be transparent enough for them to make an informed decision at which scale to execute it.

Resource provisioning belongs to two classes of distributed system issues: misconfigurations and performance issues. An initial resource provisioning may be provided when a distributed system or application is set up. If this initial provisioning is poorly chosen, the user has misconfigured their system (causing a misconfiguration issue). In autoscaling applications which request more resources or release excess ones, the resources may not properly scale to meet the demand at runtime. This is a performance issue. We investigate the problem of resource provisioning in a master-worker framework, a specific type of distributed system.

We find that there is some logical ideal scale of a distributed system. This is the scale at which some key component(s) become saturated. Adding more resources to the system would over-subscribe the key component(s). Providing fewer than that

ideal number of resources would under-subscribe the system, throttling how quickly it is able to get work done.

This ideal can change over time, especially in heterogeneous workloads, when different pieces of work (such as a job, task, or process) behave differently. We call this metric the *capacity* of the system and provide a means of calculating it while a system runs. This metric can then be used in the live system to scale up or scale down the number of resources available to that system, giving it only as much as it can effectively utilize. This approach is called *right-sizing* the system. We show the application of this right-sizing capacity model to a live master-worker system.

Beyond the capacity model, we also uncovered a lack of transparency about the state of the system. We provide two means of troubleshooting this resource provisioning issue within the master-worker framework: a web troubleshooting dashboard which ran in an open system as well as more robust, transparent logging of resource needs [66]. The term *application* in this work is somewhat synonymous with the term *system* since in this particular case (the Work Queue master-worker framework) running a research application includes an automated setup stage of the system upon which the application is executed.

We provide this work on the capacity model for master-worker applications and the problem it helps resolve as an introduction to a type of common distributed system issue and how greater transparency of the system's behavior helps resolve it. This further demonstrates, beyond a notional example, how problems which do not result in outright failure can have real, significant impacts upon a distributed system. TLQ (examined in detail in Chapter 6) expands upon this work by being applied to distributed system troubleshooting in general, provides interactive querying capabilities of the system at runtime, and ensures short-lived components (a common occurrence in master-worker applications) have their debug logs maintained for closer inspection.

3.1 Problem Introduction

Researchers have come to rely on clusters, clouds, and grids to analyze and collect data on a large scale. However, it is difficult to know the resource requirements of an application at scale. Decisions about just how large the application should be scaled often have to be made by the researcher. This can lead to cases of requesting too few resources to get their work done in a timely manner or asking for too many resources and blocking *other* researchers from getting their work done.

When executing distributed applications, users will often under-provision or over-provision their work by orders of magnitude. For example, users request resources on ten cores for an application that should be using thousands. This prevents them from getting their research done as quickly as it should. Conversely, users also request a thousand cores when only tens could be used effectively. This is a problem for the cluster since other users have to wait in the queue while their peer is holding onto almost a thousand completely idle cores. In this case, the productivity of an entire cluster can grind to a halt without intervention from a system administrator.

In principle, a user could run their application multiple times with varying resources in order to discover an appropriate resource provisioning. This is not useful in cases where the data from the application does not need to be processed more than once. It is especially detrimental when the user is charged for computation such as infrastructure-as-a-service platforms. Having to re-run the application to find an appropriate resource allocation can quickly rack up cost. It also slows down the rate of their research. It would be preferable to run the distributed application only *once* to discover an appropriate number of resources and dynamically provision them throughout the application's lifetime.

We present a method for dynamically calculating the number of computational nodes which can be effectively utilized by a master-worker system. This model is called the *capacity* of the application. This method provides the benefit that the

application does not have to be rerun, and approximations of the true value of capacity are easily obtained as tasks are executed. This model prevents waste on idle resources from over-provisioning which can in turn save users money and allocation time in the case of infrastructure-as-a-service platforms. It also increases throughput if an application experiences initial under-provisioning. The capacity model was previously evaluated with four distributed applications [66].

We also present a web-based troubleshooting tool for researchers to diagnose common resource provisioning issues in their applications with the goal of providing a transparent and informative interface to help users understand the behavior of the application at run time. The capacity model, along with basic performance metrics, provide the basis for simple visualizations which make resource issues readily apparent. The model and performance metrics also inform a troubleshooting recommendation system which provides the user with actionable steps to address potential problems.

3.2 Capacity in a Master-Work Architecture

Distributed applications executing in a master-worker system was the focus of the capacity model's implementation. In a master-worker framework, a master process serves as a centralized controller of worker nodes and is responsible for coordinating workers and feeding them tasks (shown in Figure 1.1 in the notional example from Chapter 1). In fact, it is most often the only piece of the system the user can see. Workers are processes scheduled to cluster, cloud, or grid nodes which persist so long as they receive work from the master. The master submits work to be done, called tasks, along with any necessary input data for that work. After executing their current task, a worker will provide the master with any specified output. The scalability of this application framework comes from the number of workers the master can sustain given resource availability. With fewer workers, the magnitude of concurrent

work the master can achieve is decreased, but the work will still get done. The reverse is true of being able to request more workers; concurrency will increase. However, there is a limit to how many workers an application can handle.

The degree of parallelism of a master-worker application is constrained both logically (i.e. tasks depend on each other) and practically: it is not always feasible or possible to provide the resources necessary to achieve maximum concurrency. In fact, it is sometimes detrimental for the execution of the application to over-provision it in an attempt to increase its throughput. This is because a distributed system's architecture creates two intrinsic bottlenecks. First, the execution time of each computation may be limited by the hardware available. If execution time is slow, the master will spend much of its time polling the running tasks, waiting for output. The other bottleneck is I/O time. If transmitting the input and retrieving the output of a task takes longer than tasks take to execute, the master will be stuck sending and receiving data instead of sending out new tasks.

Our definition of capacity builds directly upon an idea first presented in [143] which was then derived on first principles. The presented capacity model finds its roots in the Gustafson-Barsis Law (specifically the scaled speedup model) [38] and initial work on parallel computation speedup [37]. The Gustafson-Barsis Law states for computational workloads, there can exist portions of the workload which will experience a speedup when given more resources. As more resources are provided to the workload, that portion which can experience a speedup will continue to experience greater speedup while the latency of the remainder of the workload remains the same. This idea was later applied to parallel workloads [37].

We demonstrate there is a limit to the scalability of a master-worker application due to the bottleneck of the master process. Chiefly, we consider throughput as the bottleneck of a parallel application's capacity, so we consider the factors which drive throughput: execution time and I/O time. The model we present is a formal and

more complete consideration of the problem of resource provisioning. In particular, we first note we must weight the most recently completed task heavier than the rest. This better informs us of the application's capacity at the current state of execution as opposed to an average value computed across the whole of the application's lifetime. Without this contribution, we would be missing potential to seize more resources or scale down if the application's current state indicates that is needed. We then implement the ability to scale regardless of cores being requested by the user whereas the previous model [143] assumes each task requires only a single core. We must also track the master's own think time in the model. This think time is the time needed by the master process to complete bookkeeping, manage resources, and execute any other sub-routines. These contributions allow for a wider range of useful implementation.

3.3 Capacity Model

For each parallel application, there exists some ideal minimum number of resources which, when provisioned, give the application its fastest execution time. If the application is given fewer than the ideal number of resources, it will not reach its maximum parallelism and thus run slower. If it is provisioned more than the ideal resources, the application may run slower due to overhead incurred for managing those resources. At best, providing more resources will neither speed up nor slow down the application. This will, however, be wasting those over-provisioned resources.

Figure 3.1 models the impact upon application runtime from poor resource provisioning. The application modeled in the figure consists of 100 independent, homogeneous tasks. Each task has a 100 second turnaround time. We assume in this model that there is a 1 second cost for managing each additional computational resource. The best run time for the entire application is thus 200 seconds (in the case that all tasks are run concurrently for 100 seconds with 100 seconds of cumulative resource management time). This is shown when the scale of resources reaches 100. Since we

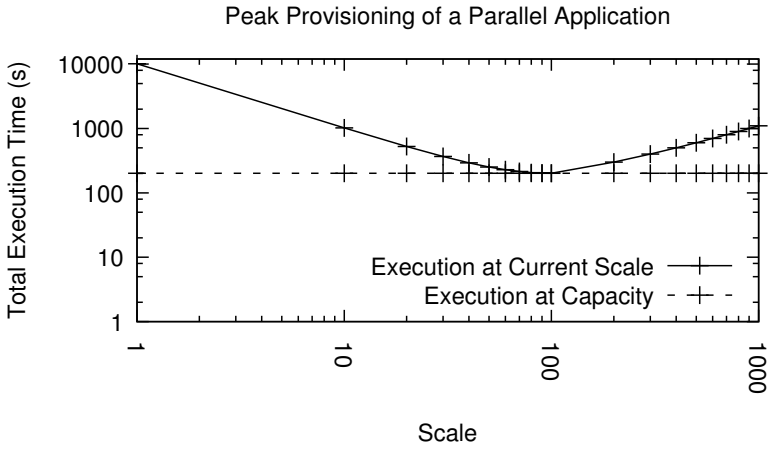

Figure 3.1. Resource provisioning impact. When executing a parallel application, there is a scale of resources which provides the fastest execution time. Provisioning fewer or greater resources will cause a slowdown.

assume there is some cost associated with managing resources, the total execution time will increase as additional resources are added after the first 100. This may not be the case for all applications, but at best adding more than the appropriate number of resources will neither increase nor decrease the execution time of the application.

We can conclude from Figure 3.1 that there is some appropriate scale for each parallel application. To provision above or below that scale will lead to a slower execution time. It would be beneficial for the user if they were able to derive some appropriate scale of resources they should request for their parallel application. We say that the application is right-sized when that appropriate scale is reached.

Now consider a master process that delivers tasks to be executed by worker processes. We can make the following observation with regards to the throughput (tasks completed per second) of the master process: assuming the master process can only transfer data (input and output) for a task at a time, all the workers have identical processing and networking capabilities, and all the tasks have identical execution time t_e and identical transfer time t_{io}, then the maximum throughput of the master

process T_m is bounded above as $T_m \leq 1/t_{io}$. The observation easily follows since the system cannot process a single task faster than t_{io}. Note that this maximum throughput is independent of the time it takes to execute a task t_e. The execution time per task t_e comes into play with the upper bound on throughput of a single worker T_w, which is bounded above as $T_w \leq 1/(t_{io}+t_e)$. If the master has C workers, their throughput upper bound, under the conditions of our observation, is CT_w.

Let C be the number of workers the master needs such that it is never idle. From the previous discussion, $CT_w = T_m$, and $C = 1 + t_e/t_{io}$. We call C the capacity of the system[1]. Since the master process deals with tasks in a sequential manner, using more workers than C will not increase the throughput of the system. The capacity C is in fact the number of workers at which a speedup curve converges [67].

3.3.1 Dynamic Capacity Model

The basic model above makes assumptions that are hard to meet in practice. For example, not all tasks are identical, and not all computing nodes have similar resources. To deal with these issues, we extend the previous computation to derive an estimate of the capacity.

Let $C_i = 1 + t_{e_i}/t_{io_i}$ be the capacity computed if all tasks *were identical* to the i^{th} finished task. Using an exponential moving average, a parameter $\alpha \in (0, 1)$ is used to weight previous completed tasks against the most recently completed one. We assume that the most recently completed task will be more indicative of the application's current behavior. In our testing, the value $\alpha = 0.05$ performed well in practice. With $C_0 = 0$, for $i > 0$ and $0 < \alpha < 1$, we recursively define:

[1]With think time t_z per task at the master, the bound on throughput becomes $1/(t_{io} + t_z)$, and the capacity is $C = (t_e + t_{io})/(t_z + t_{io})$. Here we have to be careful. If we limit the number of workers to integral values, we may find the only way to not have idle workers is to have none of them (e.g. $t_{io} + t_z > t_e$, with the floor operator giving 0). We should then execute the application locally since it cannot handle greater scale.

$$C_i = \alpha(1 + t_{e_i}/t_{io_i}) + (1 - \alpha)C_{i-1}$$

Using this dynamic model, we gain better insight into the application's resource needs. Seldom are tasks identical for an entire workload, so weighting the i^{th} finished task greater than the cumulative capacity of the previous workload allows us to better adjust when workload changes occur. For example, consider a master-worker application which has two categories of tasks: tasks of type A and tasks of type B. There are an equal number of both types. Assume that both task types have the same I/O time, but type A tasks run half as long as type B tasks. Let us also assume the workload submits all the type A tasks first then submits the B tasks. There will be a point in the workload when capacity will increase because B tasks run twice as long as A tasks. Capacity will essentially double. A more naive model [143] which does not weight the most recently completed task heavier can take awhile to adjust to the sudden change in capacity. There may be a long lag between actual capacity and realized worker acquisition. In our dynamic model, the added weight α allows our application to realize the capacity change between A and B faster. This in turn will reduce that lag and scale the number of workers quicker if the resources are available. In essence, we present a model which follows the Gustafson-Barsis Law [38, 137] to find the best speedup using an exponential moving average.

3.4 Implementation

We implemented our capacity model in the Work Queue master-worker execution framework [105]. The model can be implemented in any framework where task execution time (t_e) and task I/O time (t_{io}) are readily available. Our users run Work Queue applications to scale up their research by breaking up their analysis pipeline into smaller tasks which can be executed concurrently. Work Queue is designed to

scale from O(10) up to O(10,000) cores. The largest scale application using Work Queue has successfully scaled up to approximately 25,000 cores.

The Work Queue master is a process which the user executes on a frontend machine. It is responsible for giving workers tasks to run as well as any input files a task may require. A worker is a process which runs on a batch system and claims resources on a machine for a user's work. Worker processes persist as long as they are given work to do, and each worker has a local cache. These workers receive input files and executables to run the task if they do not have them in their cache. If the task is completed successfully, the worker waits for the master to acknowledge its success then transfers the output of the task.

The master receives a task report from a worker once a task has finished executing. If the task was completed successfully, the master uses that task report for determining capacity. This task report contains the execution time (t_e) and I/O time (t_{io}) for that task along with many other performance metrics. These metrics are measured at the worker process and are used in the calculation of capacity as shown in the model. The master also keeps track of its think time during tasks which is added to the task report. We use these times to calculate the capacity of the application, weighting the most recently completed task most heavily as defined in our model. If the task is not successful, the task report for that failed task is not included in the capacity calculation.

The master determines the capacity using the stats retrieved from successful task reports. The most recent task report's execution time, I/O time, and master think time are weighted more heavily than the rest of the application's history because we assume that task will be more indicative of the application's current behavior. After the capacity is calculated, the master submits it (and other metrics) to a catalog server which a utility called Work Queue Factory can access.

In order to request and maintain the appropriate number of workers for the ap-

plication, we use a program called Work Queue Factory to dynamically provision according to the master's capacity calculation. Work Queue Factory is an application which retrieves periodic information about a master and uses this information to submit new workers for that master. This is useful in cases where workers have idled out in an earlier section of the application but are needed at the moment. It is also helpful in scaling down by not replacing idled-out workers if the master does not need any more. The factory decides how many workers to request by calculating the minimum among the result of the capacity model, the number of tasks currently submitted by the application, and a user-defined upper bound. If the result of this calculation is less than the number of workers currently connected to the Work Queue master, the factory does not request more workers.

3.5 Capacity Model as Troubleshooting Tool

Although the capacity model provides usefulness for our users as implemented in Work Queue Factory and as a simple paper model to gauge an application's resource provisioning, we also implemented the model as the basis for a web-based troubleshooting tool. This is possible by querying a catalog server which by default every Work Queue master process communicates with in regular intervals. The catalog server is used to match workers to masters, but it also stores basic performance metrics and unique identifiers of the master.

Some of the metrics included in the catalog server are cores, memory, and disk allocated as well as how the master process is spending its time (e.g. sending input, receiving output, running application-specific code). We added capacity to these metrics. We provide these metrics in a dashboard of simple visualizations of each Work Queue master to give users insight into the behavior and current resource needs of their application. Visiting this dashboard is intended to be the first step when a user is troubleshooting resource provisioning issues with their application.

The visualizations are implemented in the D3 JavaScript library. A search function allows for a user to see all their masters on one page in the case that their Work Queue masters are sharing a pool of workers (a somewhat common tactic employed by our users). The master's metric visualizations are aligned in a tabular style to provide a straightforward dashboard for easy comparison between masters. Each data point of the visualizations provide added details on mouseover.

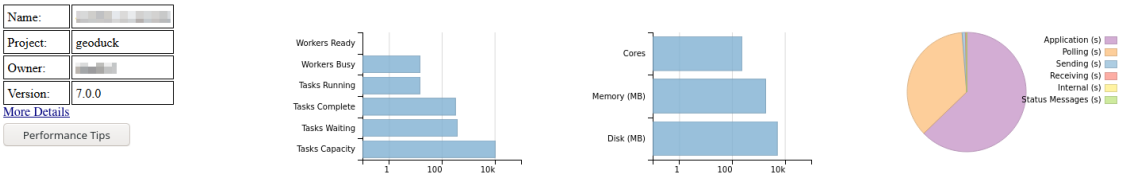

Figure 3.2. Visualization dashboard. This dashboard layout of visualizations is based on performance metrics from the central catalog server. Included are a bar chart on worker and task metrics, a bar chart on resource consumption, and a pie chart of the master's time.

In our experience, this tool has been a good first step in troubleshooting common resource provisioning issues. To make the tool more user-friendly, we have added a recommendation system which will briefly analyze a master's metrics and provide actionable steps to help the user troubleshoot. By clicking the "Performance Tips" button in Figure 3.2, the tool briefly analyzes the master metrics and produces a list of recommendations for the user. The recommendations are based on the most common factors which would contribute to a master's behavior. For example, a master which is spending much of its time polling workers (i.e. the master is idle much of the time) is most likely being under-provisioned. In this case, the recommendation system will look at how many workers the master has connected as well as its capacity and

inform the user if they would benefit from asking for more workers. Another common case is a user not providing any resources (in the form of worker processes) to their system. Not only would this be made readily transparent to the user when looking at the dashboard, they would also be informed of this when interacting with the "Performance Tips" button.

3.6 Relationship to TLQ and Open Distributed Systems Troubleshooting

With the capacity model and performance dashboard, we have resolved the issues of resource misconfiguration and runtime resource allocation as they relate to feeding a master-worker architecture with resources. However, there are many other kinds of distributed system issues. TLQ is designed to provide the framework necessary to investigate distributed systems issues *in general*.

This work introduces the catalog server as an integral part in making sense of a specific open distributed system. As Work Queue workers can come and go, can exist across multiple independent jurisdictions, and are placed on-demand when using a Work Queue Factory process, they constitute an open system of transient resources. The catalog server, in addition to advertising basic metrics about active Work Queue applications, provides a means of giving components in an open system a publicly-addressable name. In the case of the work on capacity, we are concerned only with Work Queue masters. The catalog server can be used to find the location of masters. The impact of the catalog server upon TLQ is explained in more detail later, however this initial work uncovered that the catalog server is itself both a treasure trove of debugging data as well as a means for advertising the existence and location of components in an open distributed system.

The capacity model as implemented in Work Queue can be readily used as a metric for troubleshooting in TLQ. Capacity is not only displayed on the web interface; it is also logged. As discussed in the background of master-worker applications,

the master is also most often the only process directly known to the user in their application, making it an obvious first stop when troubleshooting failures in master-worker applications. Even if a user knows the locations of their workers, it is likely they do not know how to directly access their logs, which is where TLQ can help. By exposing a class of performance issues to the user via the master's debug log, this class of problems (and their cause) is then made transparent in TLQ as well.

CHAPTER 4

USING DEBUG LOGS TO TROUBLESHOOT DISTRIBUTED SYSTEMS

Debug logs are frequently used as a means of capturing state information during system runtime which can be used to troubleshoot misbehaviors during and after the system has completed execution. Each type of log has its own format. Some are meant to be human-readable while others are structured to be interpreted by other programs like tabular spreadsheet editors. A log must contain enough information to be useful in piecing together the history of the component being logged, the important environment configurations and interactions the components made, and whether the component interacted with others in the system. Having examined a notional example and a specific system under study experiencing issues related to scaling, we turn our attention to some important yet tertiary questions:

- Why is a debug log a straightforward means of troubleshooting after the fact?
- What information is useful to store in a debug log for troubleshooting?
- How can a component's log give enough information to create links to others?

4.1 Why Debug Logs are Important

Logs are typically permanent records of the execution of a system. When a component logs an event or state change, it is typically added to an existing debug log file. Entries in a log are not usually altered once written, meaning the log may only increase in size during runtime (barring garbage collection schemes which prevent a log from becoming too large). Since logs contain a history of state changes per

component, we can use the set of debug logs created by the system as a means for recreating the events which occurred. If a misbehavior happens during runtime, and that event is logged, we can use troubleshooting tools to identify which state changes are likely to have caused or resulted from that misbehavior.

Without creating a debug log per component, the user would be responsible for actively monitoring their system for events at runtime. If they miss an indicator, and no logs are created, the provenance for that event becomes irretrievable. It will be almost as if the event did not occur. Additionally, the system under study may be running at such a scale that the user cannot effectively monitor each component.

Expecting a user (perhaps a researcher not well-versed in debugging) to act as a Panopticon, viewing each piece of their system in detail while it runs, is unrealistic in the best case. At worst case, they may find an indicator but have no means of deciphering its meaning. Keeping logs of events and state changes allows a user to provide experts with the history of their system, proving that misbehaviors did occur and where.

We could address this issue by having all relevant events and state changes be reported as messages to some centralized collector, perhaps on a machine the user can directly access. This would essentially create one debug log for the whole system. However, this approach would fall prey to a few key factors which would negate the benefit of this particular kind of logging. A performance factor to consider is the scale of messages being sent to this centralized collector. The number of components reporting (or the quantity of data being reported) may overwhelm the collector, causing important messages to either be reported with some degree of latency or not at all in the case of a buffer overflow.

Another factor to consider is the ordering of events. When logging is performed locally, each component has a consistent history of events. Creating a shared history of a system is a difficult feat to accomplish [69]. Event A may cause event B, but

what happens if the report for B shows up before A? What if the report for A is lost? This centralized log loses its usefulness. Instead, having each component create their own log is more consistent, in many cases more performant, and provides a more complete view of a system. The issue this introduces is a difficulty in *retrieving* each log, which is the thrust of TLQ's architecture (formally introduced in Chapter 6).

4.2 Common Traits Among Log Formats

No matter the format of a log (i.e. structured for human or machine readability), there are some common traits among many of them when effectively implemented. Reporting initial configuration details is often helpful for understanding the runtime environment of a component. Logging information such as local start time, the working directory/directories, the values of relevant environment variables, and known identifiers for a component (such as `jobID` for a batch system job) are important for effectively reporting a successful setup before a component begins its computation. Often, these initial values have long-reaching effects, so logging them at the beginning of a component's lifetime ensures the user is able to discover upfront the environment within which a particular component executed.

Beyond gathering the surrounding environment's state, it is also critical to report internal state changes. As the component executes, key metrics like resource usage may be reported to help a user understand the performance of the component. Progress made toward completing the component's workload are also useful since they can be investigated by the user to see if a particular piece of the workload is misbehaving. Consider a worker process within a master-worker execution framework, as was demonstrated in the notional example from Chapter 1. It may report progress in terms of certain tasks completed (referenced by a `taskID`) and whether the task completed successfully or not.

The previous two traits are critical for effectively troubleshooting a single compo-

nent. State information about the environment and a record of internal state changes help create a history of events which produced the final state of that component. A user can investigate misbehaviors on a *per component* basis this way. However, components often communicate with others in distributed systems.

Reporting interactions between components is important for establishing concrete links after the fact. Component A may send a message to component B, altering some internal state for B. However, if component B's log never reports that it received a message from A, it will be as if that interaction never occurred when inspecting the log after runtime. It is also helpful for all components involved in an interaction to record it in their own logs rather than rely on a single component to keep track. In the case that one log becomes lost or corrupted, there are multiple records of the link established.

Finally, explicit reports of errors, failures, misconfigurations, missing data, and other unexpected behavior are critical for finding issues in a component. If a component misbehaves at runtime without recording that event, it becomes harder for a user to troubleshoot the issue. They would have to rely on the reporting of environment configuration and internal state changes to piece together just what combination of environment and state caused the misbehavior. It is preferable for a component to explicitly report encountered failures (including whether they are fatal) or misbehaviors. Ideally, these reports are logged with or near any relevant log messages about the state of the component when it experienced the issue.

4.3 Recording Links Between Components

Not all records of links between components are the same. There are two degrees to which component interactions may be reported: implicit links and explicit links. Implicit links are less helpful while still providing a user with more information than *not* reporting. Explicit links, on the other hand, provide more specific, concrete

details about component interactions.

Two components are said to be implicitly linked if a provided log message can be used to *discover* a link exists between two components without that message explicitly reporting the component being linked. For instance, a log may state that a message came from an IP address and port number, or it may have received some task definition from a master process.

```
{"msgID": 1, "msg": "Request received from 127.0.0.1:9000"}
...
{"msgID": 7, "msg": "Got taskID 23 from master"}
```

These messages may be responsible for altering some internal state, causing a misbehavior. However, the user is not told exactly *which* component sent this message. They are not told which component is tied to that IP address and port (which can also change over time), nor are they told *which* master process communicated with the component. The user is given enough information to search for the source, and perhaps they may be successful in collecting enough information *external* to this log in order to troubleshoot their issue. Providing more information within the log would be a preferable solution.

By contrast, explicitly linked components provide enough information to identify precisely which components interacted and how to find them. To understand a link, the user only has to read the given log. They do not need to consult external information as will often be necessary with implicit links.

```
{"msgID": 1,
  "msg": "Request received from user client at 127.0.0.1:9000"}
...
{"msgID": 7,
  "msg": "Got taskID 23 from masterID A56B at 127.0.0.1:7281"}
```

Unlike in the implicit examples, both explicit link messages contain enough information to tie two components together. In the first message, the *type* of component (in this case a user client) is identified along with an IP address and port number. Preferably, a timestamp would also be provided to further specify exactly which user client using `127.0.0.1:9000` contacted the component since an address and port combination are not necessarily unique identifiers in a system. The second message contains not only type (a master process) but also that particular master's ID and its address:port combination. Greater specificity decreases the user's reliance on finding outside information to troubleshoot their system. We demonstrate how TLQ effectively uses explicit links to make troubleshooting more approachable for users executing an open system in Chapter 6.

4.4 Key Idea of TLQ using Debug Logs

TLQ is an architecture for troubleshooting open distributed systems. It enables effective troubleshooting by exposing the existence and location of debug logs to a user. By knowing where a log exists, a user is able to directly query it in place or selectively retrieve it rather than setting up a complicated mechanism for centrally collecting all their system's debug output.

Additionally, each debug log is parsed and transformed into a uniform JSON representation no matter the original format of the log. Knowing both how to access a log and its format, the user can then submit the same queries across all logs. These queries are used to narrow down the user's investigation of misbehaviors in their system, expose important metadata to the user, and provide a means of retrieving any relevant log to the user's machine.

Debug logs are the data produced by the system TLQ requires to operate. Since each log represents a history of configuration, state changes, and events of a component, they are necessary to keep in order to effectively troubleshoot components both

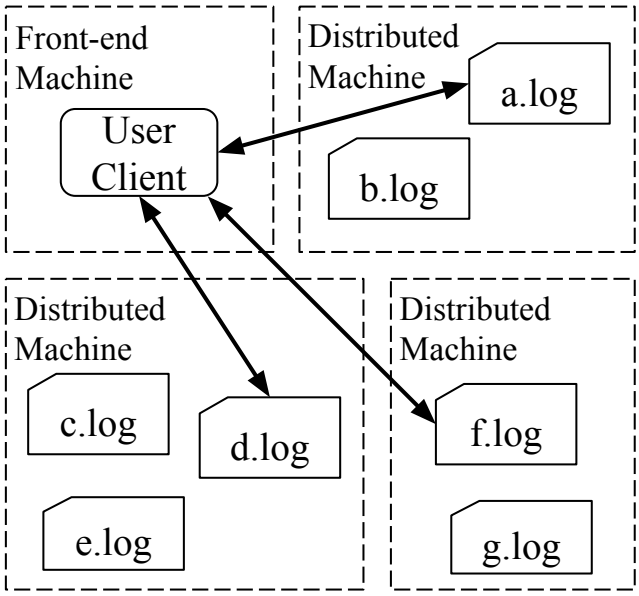

Figure 4.1. TLQ key idea. The existence and location of each debug log is exposed to a user client. The client is able to directly query and retrieve debug logs existing on distributed machines in their open system.

at runtime and after the fact. A debug log proves the existence of a component in the system and establishes its lifetime relative to the machine on which it ran. It is a single, readable, and queryable artifact of a component's execution. As such, it is an invaluable resource. TLQ focuses on providing a user direct access to their logs in an open system for these reasons; they are the critical puzzle pieces for performing useful troubleshooting.

As discussed in Chapter 1, discovering where logs exist and making them available are not trivial challenges, especially in open systems. The key idea of TLQ is that the user client should be able to directly interact with their system's produced debug logs *where they exist* rather than try to collect all these logs in one centralized location (an at best impractical and at worst impossible proposal). The TLQ architecture provides mechanisms which make this level of interaction possible, enforces unique names for each debug log in the system, and implements powerful, expressive querying

on top of allowing a user to continue running typical command line troubleshooting tools on their remote logs. Figure 4.1 shows the desired functionality of TLQ: a user directly interacting with and accessing their logs where they were created. We present the key performance considerations of the mechanisms fundamental to TLQ, the basic structure of the TLQ architecture, the importance of choosing an effective query language, and finally the querying mechanism of TLQ.

CHAPTER 5

TRACING OVERHEAD AND SCALABILITY OF KEY MECHANISMS

The primary results of TLQ are derived from its core functionality: log discovery, log custody, and the web-inspired approach to querying. The effectiveness and correctness of these functionalities are demonstrated in Chapters 6 and 8. Before delving into TLQ, we must lay the foundation of some key performance considerations. Namely those are the cost of tracing a component if it does not produce its own log, the scalability of a server keeping track of many debug logs (a key piece of the TLQ architecture), and the scalability of a user client retrieving data.

5.1 Overhead of System Call Tracing as a Debug Log

If a component does not produce its own debug log, there will be precious few records that the component ever existed. Further, those few records will probably not be able to tell us anything interesting about what that component *did* and how it interacted with its compute environment. In order to effectively troubleshoot issues in a distributed system, each component must produce some form of meaningful debug output.

If no log is created, we can use a tracing utility to create one for it. We demonstrate how `ltrace` can be used to fill this logging capability. This can be used to capture the component's system level environment interactions and system level failures. Although explicit logging by the component may provide more domain-specific context than the more generic system error messages, it provides fine-grained details of its execution all the same. However, this comes at a cost. System calls

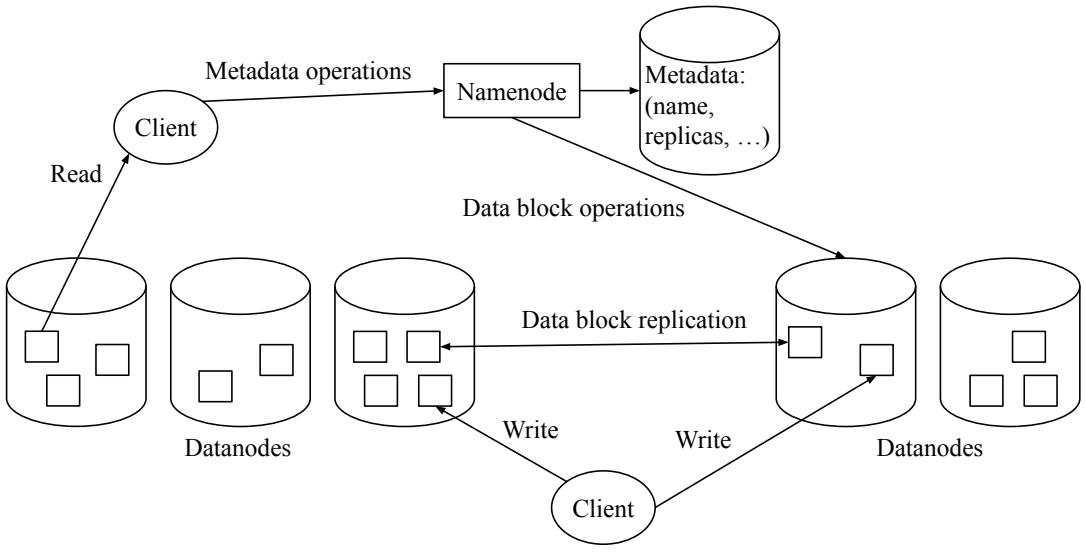

Figure 5.1. HDFS architecture. Client processes communicate with the namenode to retrieve file metadata. The client can interact with the file stored at a datanode once given the necessary metadata from the namenode. Data blocks written to datanodes are replicated.

are intercepted, logged, and passed through to the operating system. This increases the overall execution time of the component. We demonstrate this overhead with three utilities: the Hadoop filesystem's `ls`, a scientific workflow process starting up a concurrent master-worker framework master process and running a workflow, and initializing Matlab.

The Hadoop filesystem (HDFS) [116] is itself a distributed system. There is a centralized namenode which performs metadata operations on the filesystem and determines the placement of data blocks and replicas. Data is replicated across multiple machines connected to HDFS. Performing HDFS' `ls` operation not only has many metadata operations like the operating system level command, it also participates (albeit briefly) in the Hadoop distributed ecosystem. Figure 5.1 demonstrates the HDFS architecture.

This simple operation has complexity in the backend, which is largely invisible

to the user. Running `ltrace` captures many of these interactions which may be relevant for troubleshooting. For example, executing this seemingly simple `ls` causes HDFS to spin off multiple Java processes which in turn each instantiate multiple threads. Some of these Java processes encountered an internal Java error, resulting in their early termination. However, HDFS recovered from these errors without being transparent with the user these processes were terminating. Using `ltrace` revealed this hidden issue. TLQ (formally presented in Chapter 6) would allow a user to more easily uncover this opaque behavior.

We configured a recursive HDFS `ls` over 20 directories and $65,500$ files. Without tracing, the command executed in 11.80 seconds on average. With tracing enabled, it completed in 15.48 seconds. At the $O(10,000)$ scale, an overhead of 31.18% is significant. We performed the command non-recursively (i.e. only returning a list of the 20 top-level directories). The overhead for this execution was 34.79% on average (5.32s untraced compared with 7.17s traced), roughly matching the overhead at the larger scale. Compared to the operating system level `ls`, we see the logging overhead introduced is due to HDFS' moving parts. Without tracing, running traditional `ls` over the same set of files took on average 0.45 seconds while with `ltrace` enabled took 0.48 seconds (an overhead of 6.66%). All things being equal, this roughly 33% overhead on average is the price paid for tracing HDFS' `ls` operation which must be paid if troubleshooting HDFS operations since general filesystem commands in HDFS do not have options to log any debug output.

We also demonstrate the Makeflow workflow management system and the Work Queue master-worker framework's master process being traced while executing a brief workflow. The architecture for this system was introduced in Figure 1.1 for our initial notional example. When using Work Queue as its execution engine, Makeflow and the Work Queue master process are co-located (and are often treated as one logical component though they are in reality two distinct components).

Makeflow executed a workflow consisting of 1,887 tasks. The Makeflow and Work Queue master processes were the only things traced (not the individual tasks). We provided 30 workers to execute it. Without `ltrace`, the workflow completed in roughly 5.35 minutes (321.26 seconds). This short execution time is due to every Java task (a significant part of the work) critically failing early in their respective executions. The reason for this failure is uncovered in Chapter 9.

When Makeflow and the master process were traced, the total runtime (using the *same* worker processes) was 5.90 minutes (354.26 seconds). This led to an overhead of 10.30% in the execution of the whole workflow. Although this workflow involved plenty of distributed communication, like in the HDFS example, we note using `ltrace` on Makeflow and the master process led to relatively low overhead due to the kind of work being done. Makeflow is in charge of submitting work and input data (via the Work Queue master) to remote workers and receiving output from those workers. When it is not doing either of these two actions, it is either performing bookkeeping or simply waiting.

There are some applications where this added logging has a crippling overhead. Matlab is a program which is both popular among researchers and loads many libraries and files during its execution. It is an ideal application to demonstrate the extreme end of `ltrace`'s effect on application runtime. We ran Matlab and configured it to call `exit(0)` upon successful loading. Although this may seem trivial at face value, we chose to simply start Matlab because of the number of libraries and files it accesses on startup. Without `ltrace`, it took 12.01 seconds to start and exit Matlab. With `ltrace`, however, it took 5 minutes (312.41 seconds on average). We can conclude that it can be very expensive to turn on all debugging options if the application accesses many files (approximately 2,000 in Matlab's startup). Depending on the depth of logging needed to properly troubleshoot the system, this cost may be unavoidable.

5.2 Scalability of Log Servers

Consider a server which is in charge of two actions: parsing a set of debug logs into a JSON representation and responding to requests from clients to download logs from that set. We discuss in Chapter 6 how these two actions are key mechanisms for TLQ. We first evaluate the scalability of these foundational concepts.

A log server has two inherent factors limiting its scalability: the time taken to parse logs and the storage space required for both debugs log and the parsed JSON. We first demonstrate how parsing time can render a log server unusable at certain scales. We then explain how the log server's storage needs may be a concern for those with limited disk space (which may rapidly become exhausted by the server).

5.2.1 Parsing Stored Data

One of the most time consuming actions a log server performs is parsing debug logs into JSON. In order to prevent blocking at the server, consider this periodic parsing is executed on a thread. When a new log is added or an existing log is modified, it is (re)parsed by this thread.

We measured the overhead of this parsing using logs of uniform size (but not uniform content). A parser was written to process these logs, transforming each line into a JSON representation and adding some metadata about the file in a top-level JSON object. The data stored at the log server was increased linearly. Each log was 100MB in size and was approximately 68,000 lines long. We captured the time taken to parse the entirety of all existing logs at different scales and noticed a linear increase in the time taken to parse all logs. Table 5.1 shows this linear increase.

If the log server blocked while parsing, no client requests would be answered once a certain scale of debug log data had been stored. Creating a thread to be responsible for the parsing solves only half the problem. At a certain scale, the parsing would essentially be constantly happening in the background. If a user requested some data

TABLE 5.1

LOG SERVER PARSE TIME.

Logs	Total Size	Total Parse Time
1	100MB	4.87s
10	1GB	45.29s
100	10GB	455.59s
1,000	100GB	4,546.88s

which had either not yet been (re)parsed or was in the process of being parsed, the user would get out of date information about that log (if the log had been previously parsed) or an error that the JSON version of the log does not yet exist (in the case the log had not been parsed before). It is also probable that the parsing would rapidly fall behind the outstanding logs yet to be parsed, especially if new components became actively watched by the log server. In either case, the log server can become an unreliable actor. It will be unable to provide (relatively) up-to-date information.

5.2.2 Size of Stored Data

Continuing to use the example of the previous set of logs, we also see another limit to the scalability of a log server. Disk space is not only consumed by the debug logs over which the log server takes custody. The parsed JSON logs are another significant source of disk usage.

Each JSON log is at least the same size of the raw debug log. This is because each line of the log is transformed into a JSON representation, and additional JSON metadata may be added to the log (which is utilized by TLQ to great effect). If a log server is started on a machine with low available disk, the server may quickly exhaust

the machine's available disk space even if the machine can store the raw debug logs. Even in machines with a vast amount of available space, the log server may be deemed to be consuming too much for a user's disk allocation (perhaps breaking some system administrator's policies). If the server is running in an infrastructure-as-a-service platform, its disk consumption behavior may become costly. This is especially true if the user budgeted only for the total anticipated disk usage of their system without adding in the storage overhead for the parsed JSON versions of each log.

One alternative that comes to mind is deleting unused or inactive log data (and perhaps their respective JSON logs). A server can handle individual delete requests from a user, but assume instead it chose to automatically perform garbage collection periodically. If the garbage collection window is too short, the benefits to a user are negated. They may not ask for a debug log until after the log server has already deleted it. Ephemeral components which have a very short lifetime may have their logs evicted before the user decides to query them, effectively bringing us back to the initial issue of ephemeral components: blink and you may miss them, taking all evidence of their existence with them. If the window is too large, the server may not evict data quickly enough and fill the disk (putting us in the same position as if we did not evict at all).

Since the behavior of distributed systems vary so drastically between each other, it is rational to have each log server retain both debug logs and the JSON logs until a user explicitly tells them to evict the data. In some systems, there may be components which run for hours or days. This necessitates that their debug logs be at least as equally long-lived. Ephemeral components (existing on the order of minutes, seconds, or less) need only have their logs exist as long as they are relevant to the user. Perhaps their impact on the system as a whole is relegated to other components which share some temporal locality with them (e.g. components which existed shortly before, at the same time as, or shortly after the ephemeral component). Or, perhaps

their computation has long-running implications which stretch until the end of the system's execution. It is impossible to know this at runtime without having significant domain knowledge about the system under study, so it is reasonable that log servers not evict data until told to do so.

5.3 Scalability of a User Client

We have established there exist log servers which parse debug logs and allow users to download logs local to that server. Now we turn our attention to the scalability of a user client requesting those downloads. The client performs queries upon debug logs, pulling relevant information out of them. The relevance of this action to troubleshooting a distributed system is apparent. We provide results for TLQ's querying mechanism (discussed in detail in Chapter 8.1), however the same concept may be applied to many query languages and evaluation mechanisms.

The primary limit to scalability at the user client is the overhead of performing queries. A query is resolved in three stages: fetch all needed data, evaluate query expression(s), and return the result. Fetching involves contacting a log server and downloading a log to the client's working directory. Query evaluation time depends on the query language and evaluation mechanism used. Fetching data and evaluating the query upon that data are the two stages which act as the bottleneck to making progress at the client. At a certain scale, queries become long-lived computations. This is not unreasonable since running a query on very large data should be expected to take a long time, however it may become inconvenient to the user to evaluate a query at that scale. The user may instead opt to use command line tools or arbitrarily reduce the size of their fetches to perform queries in chunks.

TABLE 5.2

USER CLIENT QUERY TIME.

Logs	Total Size	Fetch Time	Total Query Time
1	100MB	3.54s	5.60s
10	1GB	35.79s	55.87s
100	10GB	385.54s	596.31s

5.3.1 Fetching and Evaluating Queried Data

Using the same data generated to highlight the performance limits of the log server, we demonstrate the scale at which using a querying mechanism becomes inconvenient. Table 5.2 shows the number of logs fetched by the user client, the time taken to completely fetch the data, and the time taken to evaluate a query. We use TLQ's query language (called JX) further detailed in Chapter 7 to ask the question of the logs fetched: In each log, how many error messages were recorded? We end Table 5.2 at a scale of 100 logs. Fetching all $1,000$ logs from the log server (which stores the intermediate result in memory) would exceed the amount of memory on the user's machine (32GB in our case).

We note that the number of logs is a less important factor than the *size* of each log when evaluating queries. In Chapter 8.1 we demonstrate the JX language effectively querying $1,850$ logs with a total size of 278.54MB. However, in this case, each of our logs is 100MB in size. This relatively large log size puts a greater strain on the query evaluation mechanism.

Waiting approximately 6 seconds for a query to evaluate on a log 100MB in size may be quite reasonable for a user. However, we see that asking for multiple, large logs can cause queries to take quite a long time to fully resolve. While this increase

in time is linear, it does not provide much of an interactive, snappy experience for the user. If the user needs to query multiple large logs, they may be better served using lower level utilities than full-blown query languages. However, the user may lose useful functionality desired to do meaningful troubleshooting.

CHAPTER 6

LOG DISCOVERY AND LOG CUSTODY: THE FOUNDATION OF TLQ

So far we have demonstrated that troubleshooting a distributed system can be an incredibly complex task. With the work on the capacity model, we showed it is rarely feasible to expect a user to know the fine-grained interactions between their system and the environment configuration of each machine used in the system. Because of this lack of transparency, work can grind to a halt when a seemingly trivial detail changes. To address this, there is a plethora of state-of-the-art log analysis tools, debuggers, and visualization suites (many of which we noted in Chapter 2). However, as we have alluded to previously, a user may be executing in an *open* distributed system where the placement of their components are not known before runtime. This makes the process of tracking debug logs almost as difficult as troubleshooting the runtime issues these logs have recorded because the location of those logs are not typically transparent to the user (and by association the troubleshooting tools they are using).

To solve the transparency issue, we need a mechanism for log discovery to make the existence of debug logs known to the user. Beyond that, we also need log custody which empowers the mechanism providing log discovery to also take ownership of debug logs and become the authoritative source of that data. In Chapter 5 we evaluated the foundational actions of these mechanisms, now we introduce them formally. TLQ (Troubleshooting via Log Query) is a framework designed to provide both these services for troubleshooting open distributed systems.

TLQ consists of a querying client and a set of servers which track relevant debug

logs spread across an open distributed system. Through a series of examples, we demonstrate how TLQ enables users to discover the locations of their system's debug logs and in turn use well-defined troubleshooting tools upon those logs in a distributed fashion. Both of these tasks were previously impractical to ask of an open distributed system without significant *a priori* knowledge. We also verify TLQ's effectiveness by way of a production system: a biodiversity scientific workflow. We note the potential storage and performance overheads of TLQ compared to a centralized, closed system approach [65].

6.1 Problem Introduction

As computational research on complex distributed systems has more rapidly become commonplace, users have experienced growing pains. Scaling up computations and deploying large-scale systems necessitates troubleshooting misbehaviors at scale, especially considering some faulty behaviors may not present themselves when executing on a single machine or at a small scale. Compared to troubleshooting an unexpected issue on their workstation, it can be much more difficult and intimidating to troubleshoot them in a large-scale distributed system. There exist plenty of troubleshooting tools which can perform log analysis, make connections between disparate components, and provide querying capability to databases which ingest the various debug logs from the system. However, these tools are rendered useless if the user is not made transparently aware *where* the debug output of each component (each service, computational unit, etc.) of their system exists, *how* to make sense of each log, and *how* each component may impact the execution of others.

To provide further complication, the user may be executing in an *open* distributed system. We define an open distributed system as a set of computing resources whose membership in a cluster, cloud, or grid may not be permanent, which are assigned computations at runtime (i.e. the system *and* user do not know in advance which com-

putations will be scheduled where), and may consist of cross-domain resources (e.g. resources coming from multiple cloud providers, campus clusters, and national-scale infrastructures) which span *independent* organizational jurisdictions. It is commonplace for a user[1] to have direct knowledge of only a single component of their system which is user-facing, however the other components of their system (e.g. worker processes, remote services, replica storage) do not need to have a fixed location at runtime. Instead, it is up to the scheduler of the underlying cluster, cloud, or grid to determine the placement of these computations. Between executions of a system (and *during* its execution), the underlying resources may vary as machines are added or removed from membership. In addition, a system may have active components communicating across domains which are under different jurisdictions (e.g. running and interacting concurrently on two clusters at different research institutions).

We have encountered a key obstacle preventing straightforward troubleshooting of open distributed systems: the user must be notified where their components land in an open system and be given a unique name which can be used to access the debug output of those components. A contemporary state-of-the-art approach is to require debug output be collected at some centralized, highly performant rendezvous point which we have seen applied to tools using the Elastic Stack (formerly ELK Stack) [68, 129, 142]. However, this approach is not ideal for *open* distributed systems since it requires the user's knowledge *a priori* what debug output will be relevant and only applies for a single domain (whereas an open distributed system may be composed of multiple domains). This state-of-the-art approach is fantastic for large, cohesive organizations such as businesses or standalone clusters, however it falls short when the system under test crosses barriers between domains (i.e. the system lives within and between the jurisdictions of multiple organizations).

[1]We define *users* as researchers accessing distributed resources to set up their own distributed systems. However, these difficulties are also applicable for system administrators, developers, or other support *assisting* these users.

From this observation we have designed TLQ, a framework for log discovery and log custody of open distributed systems. It addresses the need to provide the user a name and location for their components and debug output, and it emphasizes leaving debug output *in place* rather than collecting it all at a single, centralized machine since that approach is at best infeasible and at worst impossible for open distributed systems. TLQ's architecture allows troubleshooting tools to be placed atop its software stack to provide users the troubleshooting experience they expect albeit in a distributed fashion. Broadly, TLQ's architecture consists of two parts: a user-end querying client and a set of log watch servers which live on the open system. The user client submits calls to troubleshooting tools to be performed on logs tracked by the log watch servers. We demonstrate open distributed system troubleshooting of a biodiversity scientific workflow called Lifemapper. We also briefly discuss the overhead of this distributed approach as it applies to the troubleshooting experience. We conclude by providing three critical lessons learned about distributed systems troubleshooting which became apparent after implementing this iteration of TLQ.

6.2 Troubleshooting as Distributed Querying

Our scientific workflow notional example in Chapter 1 highlights a few key insights as to why troubleshooting distributed systems is so difficult:

- There are many logs located on many machines.
- Misbehaviors are not restricted to a single component.
- Often no single log gives *all* the context for misbehaviors.

In the notional example in Chapter 1 we assumed the user knew where their logs were located (or perhaps they were explicitly transferred to a front-end machine for manual troubleshooting). However, this is not always the case. The underlying components of the system may know where their logs are located, but this information is probably

not transparent to the user. They may not know where their logs are located let alone how to retrieve them, or perhaps it is too expensive to transfer all the logs to a centralized node. Further, transferring to a centralized node may be impossible if the system executes across multiple domains, each with their own jurisdiction (e.g. a private cloud provider and a research institution's cluster).

In the notional example from Chapter 1, the user had to read three different *types* of logs: a workflow manager log, a worker log, and multiple task logs. Either they or the tools they used had to understand the format of each log type in order to comprehend the context each log presented toward finding the cause of a misbehavior in their system. Further, each log only provided pieces of the cause of the misbehavior until the user found the misbehaving task's log. This becomes an issue of finding the needle in the haystack as the scale of distributed systems continues to increase.

Each of these insights translate into underlying problems with troubleshooting distributed systems as they continue to become the commonplace method for research and industry computing: quantity of debug output scale, understanding relationships between components, and discoverability of debug output. Each of these problems can be addressed by providing a unique name for each log, advertising it to the user, creating a more readily queryable set of metadata about each log, and applying the user's troubleshooting tools in a distributed matter which sends computation (i.e. the query) to the data rather than the other way around. We introduce TLQ, a system which keeps debug output in place (removing the need to transfer a large degree of data to a centralized node), allows for relationships between components to be more transparent (so it is easier for the user to discover these connections) by parsing out metadata about each log, and tracks where debug logs are located (leaving the user free to troubleshoot their system issues *without* having to directly know which computations happened where).

6.2.1 Querying Logs in Place Across Domains

In TLQ, we implement a service on each node of the compute resource (i.e. machines in a cluster, cloud, or grid). This service is called a log watch server, and it is informed which logs it should track by the various components of a distributed system under study. In TLQ, each component is programatically wrapped by a simple monitor script which tells the log watcher the names of the logs that component will create. This same script also reports back to the host which submitted the component (this is usually a front-end node of a cluster, cloud, or grid in our day-to-day experience with users) to tell the user where it landed and how to query that log in the future. The log watcher then periodically parses its tracked logs to draw out metadata about each log and places it as a separate JSON document which acts in part as a *metalog* of the environment of that component (the files, processes, and environment variables in the log) and to summarize the high-level status information about that component such as exit status and total runtime. This provides an at-a-glance view of each component which helps lower the noise of the total debug log output of a distributed system.

There are multiple log watch servers monitoring the distributed system's logs, each representing a fraction of the collective debug output of the whole distributed system which can then be queried by the user's choice of troubleshooting tools. This is done through a user-end client which dispatches either a query on the servers' tracked logs or a request to transfer a specific log for manual inspection. The set of log watchers are discoverable entry points for an open system since their existence is advertised to the user. Figure 6.1 demonstrates the architecture of TLQ in action.

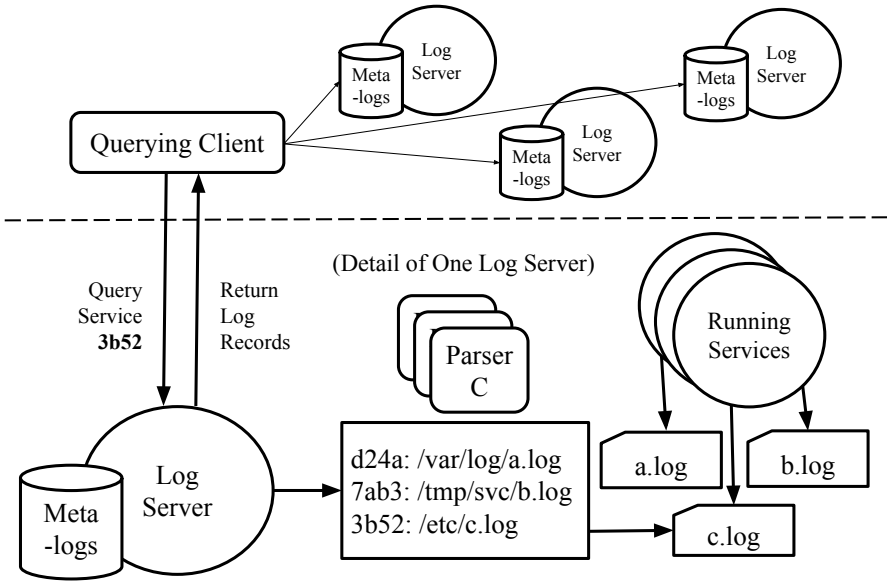

Figure 6.1. TLQ system architecture. TLQ queries are either invocations of a troubleshooting tool or a request for a particular log. Each log server manages a set of parsers that consume local log files for key information and export it to JSON documents to facilitate troubleshooting.

6.3 Implementation

The underlying troubleshooting system architecture is composed of four parts: a log watch server, a set of log parsers, a querying client, and a monitor script which communicates with the log watcher and user client. Refer to Figure 6.1 for an expanded view of this architecture. The server is designed to provide minimal HTTP communication capabilities. Its primary function is to monitor and parse logs it is told to watch and store the parsed content as JSON documents. These documents contain metadata about the raw logs they represent such as the exit status, the command executed, the files accessed, etc. In addition, each log's filename is transformed into a universally unique ID (UUID) and placed into the server's working directory when possible (thus implementing log custody). This UUID is used to provide a URL for that log and a URL corresponding to its parsed JSON document, giving each

log a unique name in the system. Two logs may have the same logical name (e.g. `debug.log`), but the log is assigned a UUID and a URL to uniquely identify it even when logical names may collide. This functionality makes it a discoverable, queryable node in an open system.

Each TLQ log watch server uses parsers to periodically add new records to its JSON documents. Each parser is designed to produce records for a specific log type (e.g. `ltrace-parser` is responsible for parsing logs created by `ltrace` to extract the processes, files, and environment variables of the component). The specific details for each parser are largely unimportant to understanding the broader architecture. We provide a set of parsers specific to the tools used in this work, but it is expected other developers provide the log parsers for their own software the goal being the developer knows best how to interpret their software's logs. TLQ needs some way to transform raw debug output into records, in order to give users a higher level view of their components' logging, and we have chosen to create parsers for that task.

The querying client provides the user the capability to use the troubleshooting tools of their choice upon specific components across their system. In this work, we demonstrate how `grep` can be used to troubleshoot an open distributed system. In order to use these tools, the client must be made aware of which logs it can query. The list of logs available to the client, represented as UUIDs, is accumulated by a local lightweight server running alongside the client. A monitor script is used to execute each component of the system, and it communicates with both its respective log watch server (on whichever machine the component lands) and with the client-end server to ensure both parties know the UUID for the component being monitored.

The monitor shell script writes to a file in the log watch server's working directory. Each line it writes describes a log the server should watch and includes the name of the distributed system to which the log belongs, the absolute path to the log (which may be user specified or a defined value by the component's code), and the component

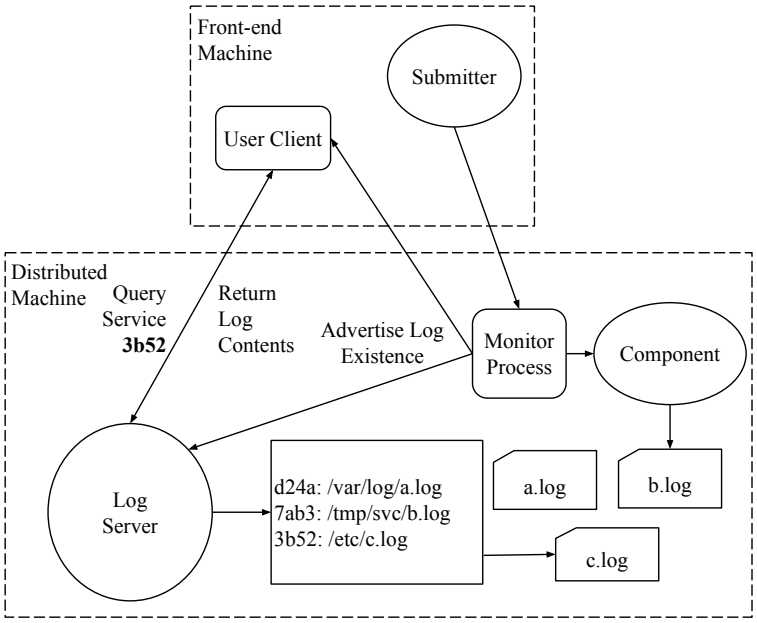

Figure 6.2. Monitor operation. Each component is wrapped by a monitor script on submission. The monitor advertises the log created by the component to its local log server and to a user client.

type(s) contained in the log. The server periodically checks this file, updates its list of files to watch, and writes this list to a separate file which includes a unique ID used to identify the debug file at the client-side. The monitor script waits for the server to update this list with the IDs of the files the monitor told the server to watch. The script then reports back to the client the system name, host and port of the server, and relevant file IDs. The monitor then executes the command it had wrapped, starting up a component of the system. Figure 6.2 demonstrates the monitor in action.

6.4 Evaluation

We demonstrate the effectiveness of TLQ for facilitating open distributed system troubleshooting by utilizing the popular tool `grep` to diagnose intermittent failures

encountered while executing the Lifemapper biodiversity workflow across two separate administrative domains (in this case, two separate campus-scale clusters). Lifemapper is a biodiversity scientific workflow [114] executed using the Makeflow workflow management system [127]. The Lifemapper project[2] is designed to create a species biodiversity map of the world. The workflow is an instance of this larger project which takes an input dataset of georeferenced biological samples and correlates them to certain environmental models.

The input dataset for Lifemapper contained some unexpected and invalid records. Tasks which consumed this improper data would explicitly fail, leading to a segmentation fault. This was a source of frustration while designing the workflow. We knew before running TLQ that this issue existed, and we demonstrate that TLQ makes it convenient to verify this known issue.

TLQ provides the log discovery mechanism and the capability to run `grep` at each log watcher by way of the user client. We show that, at Lifemapper's scale, distributed queries with TLQ perform on par with the collect-and-query approach utilized by centralized architectures. We further demonstrate, by way of model, the scale at which distributed querying can outperform collect-and-query given certain conditions.

Makeflows consist of a set of rules, like a Makefile in GNU Make. Each rule contains a set of inputs, a set of expected outputs, and a command which utilizes the inputs to create the outputs. Through these rules, Makeflow creates a directed acyclic graph (DAG) of data dependencies which determine both the parallelism of the workflow and whether the workflow is complete (i.e. when the final outputs are created). Figure 6.3 show the DAG structure of Lifemapper at a small scale. The Lifemapper workflow executed in this work consists of $1,887$ rules, and it has 655MB of input data. It took approximately 45 minutes to complete the workflow from start

[2]Lifemapper project found at: `https://lifemapper.ku.edu/`

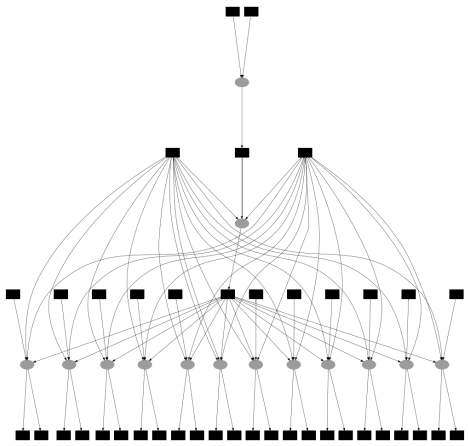

Figure 6.3. Lifemapper structure. Squares represent files, and circles represent processes. Processes are dependent upon input files and produce output files. Its structure allows for a high degree of parallelism.

to finish.

We made use of the Work Queue master-worker framework to run the Makeflow rules. Figure 1.1 from the notional example demonstrates how Makeflow and Work Queue interoperate. A single master process accepts rules from Makeflow and submits them to connected workers as tasks. Makeflow rules and Work Queue tasks are roughly equivalent definitions of work to be done. Each worker executes on a separate machine from the master and transfers input and output data to and from the master node. Each worker also has its own data cache to avoid unnecessary duplicate transfers.

The worker processes were submitted as pilot jobs to the HTCondor batch system, which scheduled the workers onto their respective machines. In all, the distributed system set up to run Lifemapper consists of five types of components: the workflow management system, the master process, worker processes, the batch system interface, and the actual computations of Lifemapper (the commands executed in each Makeflow rule). This creates a hierarchy of communication which, at scale, can

become increasingly difficult to troubleshoot when issues arise. The workflow management system, master process, and batch system interface logs are all created at the same node the user accesses, so TLQ does not need to discover and advertise these logs to the user. The worker logs and traces of the tasks, however, *do* need TLQ's help to become transparent to the user. We used 15 workers to execute Lifemapper (a reasonable scale given the ability of the master to feed work to its workers [66]).

Approximately two thirds of Lifemapper's rules execute Java code (the other third being Python). Throughout Lifemapper's runtime, the system encountered intermittent unhandled exceptions (`java.util.NoSuchElementException`) which led to a number of rules producing incomplete output. We ran Lifemapper five times to confirm these failures were intermittent and not a baked-in failure due to the workflow's construction. These failures became apparent from the `STDOUT` captured by the Work Queue master process (which captures forwarded console output from its workers).

From this output, we can then match the error to the command executed in Makeflow's log. From here, we look up that command in the collection of logs which the various log watchers are tracking (provided to us by the monitor). We find it, and through the TLQ client we query the relevant log using `grep`. Based on the error (and unhandled exception) we would expect that a search for `SIGSEGV` would turn up some useful results. From `STDOUT` to local higher-level logs to the raw debug logs, we have found out why certain rules failed in Lifemapper. Indeed, we find that certain rules in Lifemapper fail early (producing only partial output). We later learned this failure was due to improper input data causing the Java program to reach an error state. Multiple threads of each failed rule encountered segmentation faults due to the unhandled exception:

TABLE 6.1

LIFEMAPPER QUERY ROUNDTRIP TIME.

	Distributed Query	Collect-and-Query
One Log	0.02s	2.45s
All Logs	10.30s	2.56s

```
11026   3.629026 --- SIGSEGV (Segmentation fault) ---
11041   0.412124 --- SIGSEGV (Segmentation fault) ---
11044   0.880959 --- SIGSEGV (Segmentation fault) ---
```

Table 6.1 briefly summarizes the time taken to perform queries using TLQ and performing collect-and-query. In total, Lifemapper produced 144MB of log data remotely. These were Work Queue worker logs and traces recorded from running `ltrace` on each rule. This demonstrates the scale at which smaller workflows generate log data (roughly 22% the size of the input dataset). We see that querying only one log (to verify the Java exception resulting in a segfault) using TLQ outperforms the collect-and-query approach since this second method requires the transfer of *all* logs before queries can occur. However, given the scale of the log data, collect-and-query performs better querying all logs once it has collected them than TLQ does querying each log individually. This is due to the overhead of opening connections one-by-one rather than transferring all the files *en masse* and performing queries locally. What these results tell us is that the most common approach to troubleshooting (i.e. asking questions of just *one* log at a time) provides a performance benefit compared to the centralized collect-and-query approach. However, TLQ's performance does not scale as well as collect-and-query if the user needs to query *each* log. Conceptually,

querying all the logs via TLQ is akin to the difference between performing a `SELECT *` query upon a single, local database and performing a `SELECT *` query upon multiple, remote databases.

6.4.1 Distributed Queries at Scale

Lifemapper, while a real-world example of an open distributed system, does not produce a large degree of log output data (measuring O(100)MB) to demonstrate the performance of TLQ as compared to collect-and-query at larger scales. So, we model and discuss the effects of distributed querying versus collect-and-query to elaborate upon initial observations gleaned from TLQ in use. Specifically, we demonstrate the (in)efficiency of data transfers compared to the amount of relevant data to be queried. We also look at the impact upon system throughput at the collection node when all logs are streamed to one location.

TLQ keeps logs in place, allowing queries to be performed at each log watch server. Contemporary state-of-the-art architectures require collecting logs to a centralized node before queries can be performed. The names and locations of logs in the centralized approach must be known *before* creation thus avoiding the log discovery problem at the cost of flexibility. Figure 6.4 demonstrates the scale at which benefits of querying logs in place rather than collecting them at a centralized node become apparent. We can easily model the scalability of distributed queries against the collect-and-query approach in a way which is fair to both methods. We make the generous assumption that the network transfer speed is equivalent to the local disk speed (which may be the case for a system running in a completely local network such as a supercomputer). We also assume there is a small, fixed network communication overhead to transferring data and to performing queries. Our final assumption is that the query execution (regardless if it is done locally or distributed) will only output either 1% or 10% of the total data queried since a user typically

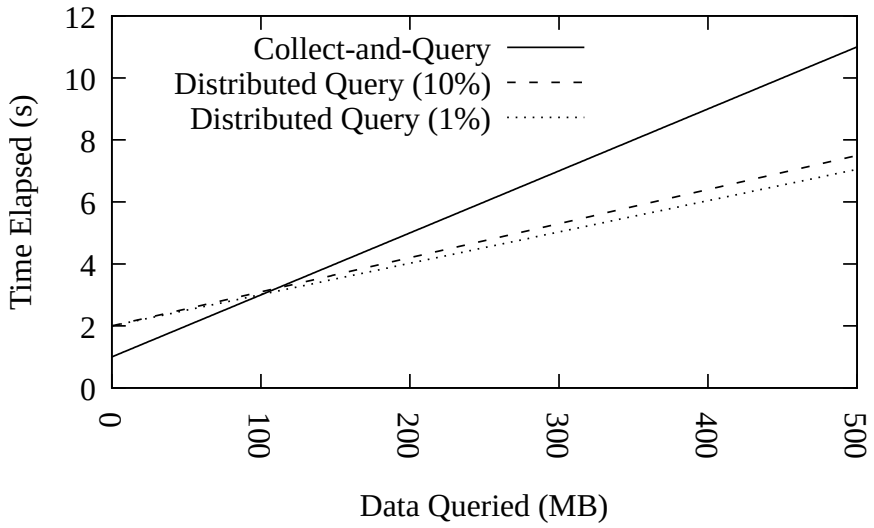

Figure 6.4. Cost of collecting and querying. All things being equal (communication overhead, transfer speed, and local read speed) there is a scale at which distributed queries are faster than the collect-and-query approach used by centralized architectures.

investigates fairly specific error messages or oddities in their logs. Without this final assumption, the distributed query time would match quite closely with the collect-and-query time since both would roughly measure the time taken to transfer (collect) and read (query) a file.

At small scale, the collect-and-query approach performs as well as distributed queries. In fact, when the scale is less that 100MB of log data as was practically the case with the Lifemapper workflow, the collect-and-query approach performs better due avoiding the communication overhead of sending over the query to the log watcher. However, as the scale continues to increase the benefit of only transferring over the relevant parts of a log becomes apparent, and we see that this crossover in performance occurs quite quickly. As stated previously, we assumed the best case scenario to the benefit of both approaches. Should network performance falter due to system load, both approaches would in turn suffer at the same rate. If local

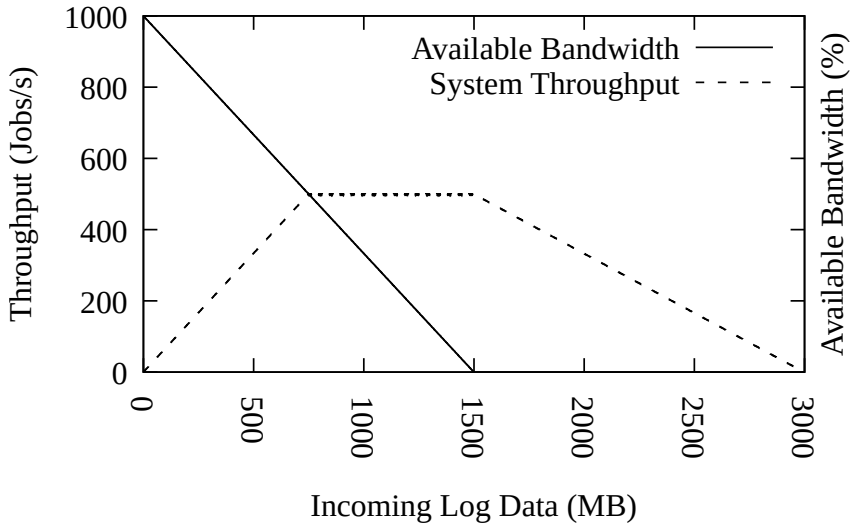

Figure 6.5. Effect of centralizing log collection. There is some scale at which centrally collecting all logs degrades system throughput due to unavailable bandwidth (i.e. the system spends so much time transmitting log and output data that it cannot do anything else).

disk performance were to slow due to system load, the collect-and-query approach would suffer most since all logs are queried locally. The same would be true for the distributed query approach is a log watcher node's disk was under heavy load.

Figure 6.5 demonstrates the effect centralizing the collection of log data has upon the distributed system under study's throughput (defined generally as jobs completed per second). This highlights an observation about centralization in open distributed systems: there is some scale at which collecting more and more data in one location will bring forward progress of the system to a halt. In this case, we denote this as system throughput (represented as jobs completed per second).

Recall that there is some maximum effective scale of a system called its capacity [66], and adding log data transfers further erodes the total capacity of a system. We see this at play when the system reaches its capacity, plateaus, and then begins to decrease in throughput as the available bandwidth of the system decreases. We

can observe this effect from two perspectives: from the node collecting the logs and from the nodes sending their logs to the centralized repository. If we centralize the collection of logs to the front-end machine (typically the machine the user accesses), Figure 6.5 demonstrates the degradation of new work being submitted *from* the front-end machine *to* other nodes in the system. Since it will be spending so much time collecting log data (along with any output data from the computations), the front-end machine cannot submit more work. It is bogged down in file transfers which fill up the bandwidth of the machine, preventing the dispatching of more work.

The converse is true when viewing Figure 6.5 from the perspective of individual nodes interacting with the front-end (or some other specified collection node). The system will still come to a halt because the nodes doing the heavy lifting (running components and computations) cannot *receive* more work because they are spending their time sending logs to a centralized node. Further, that centralized node's bandwidth will eventually be exceeded meaning the other machines will have to *wait* to transmit their log data rather than perform more work.

A *critical* caveat which must be made clear for both these models is that they assume centralized log collection is even *possible* in the first place. TLQ was designed, from first principles, contrary to this assumption. Open distributed systems can function much like the World Wide Web. They can be executed on tightly coupled, highly optimized, high locality machines like supercomputers or an organization's data center (which *do* allow for centralization of logs quite well), or they can be loosely federated, heterogeneous groups of machines out in the ether in which each group is not aware of others' existence unless those others' locations have been advertised (centralization is *not* possible here).

In the first case, TLQ is useful out of practicality. Leaving the logs in place and advertising their location is one less step the user needs to worry about when setting up their system, and when only investigating a small portion of the debug output

of a system TLQ provides some performance benefits at large scale. In the second case, TLQ and architectures like it are a necessity. They provide a mechanism for log discovery and log querying when centralization is not a practical possibility.

6.5 Three Lessons Learned

We learned three critical lessons about open distributed system troubleshooting when implementing TLQ. These pertain to how the querying experience for distributed systems troubleshooting is often only partially satisfied by current tools, how a component of a distributed system essentially provides a scope and context for computations, and how the differing structure of logs makes it difficult to connect one component to another explicitly (and introduces the need to write multiple parsers for logs).

Current tools, as demonstrated with our use of `grep`, are typically used to investigate one context at a time. This translates to one component or log in TLQ. However, we have shown that there are relationships *between* components in distributed systems. These relationships can be uncovered and investigated with *iterative* queries from the client. This would result in chaining tools together, performing successive invocations of the same tool, etc.

Our definition of a component also ended up being quite different at the end of this implementation of TLQ than it was from its outset. Thus far we have presented a component as a service or computation which is an atomic definition of work in the distributed system. It interacts with its runtime environment (i.e. files, processes, and environment variables). It is a piece of the whole system which the user submits to a compute resource. However, after implementing TLQ using this working definition we learned this was not sufficient in describing a component. Really, a component in a distributed system is a (*hopefully*) sane environment context in addition to being the unit of work for a user. We figured this out after running into a problem: names

are hard, especially in a set of uncoordinated distributed components. We cannot trust, even on the same machine, that component 1's file `/a/b/c` is equivalent to component 2's file `/a/b/c`. We must treat all the environment used by a component to be contained within the context of that component though that component may interact with others.

We found a significant pain point to implementing TLQ was the necessity to write a set of parsers able to read in multiple log formats, extract the uniquely defined records from those logs, and then have the log watch server give each of those unique records an ID. Addressing the previous lesson of how names are hard, it would be preferable to have components name themselves in some statistically unique manner, placing this name in its own log(s) front and center. Whenever its log(s) are read by the log watcher, the server knows exactly who it is dealing with. Further, when one component communicates with another, it should pass along its name which should in turn be noted in the recipient's log (e.g. "I communicated with worker ABC-123, and it sent me the following message: ..."). This would make relationships between components *concrete* rather than implied through data exploration as is done in TLQ. Further, it allows for a straightforward implementation of query chaining as discussed previously.

There would be no need for specialized parsing if logs shared a common, transactional format. This format, possibly JSON due to its ubiquity, would capture each recorded state change as a transaction listing as much information as is relevant all on one line, with keys and values defined for the log watcher. Many logs are already structured in a transactional way, so transforming existing components' logging mechanisms to a standard, transactional format would be straightforward.

CHAPTER 7

QUERY MODELS

Having explored TLQ's command line utility querying capability, we now expand to applying a proper query language to TLQ. The JX language provides the capability for the user client to fetch JSON metalogs from log servers and perform queries directly upon those logs. Rather than work within the confines of already existing troubleshooting tools and command line utilities, JX matches the format of the JSON metalogs.

We chose to expand JX to include querying capabilities after trying various existing database management technologies. In particular, we adopted SQLite (SQL), RethinkDB (NoSQL), and GraphQL (property graph traversal engine) during the early stages of TLQ's development. Each of these had issues which prevented their wholesale adoption as the querying engine within TLQ. JX, which was developed within the Cooperative Computing Lab along with Makeflow and Work Queue, was more readily adopted because we could more readily implement any functionality which we found missing from the other three querying engines.

7.1 SQLite

We first demonstrate SQLite's relational database approach to querying data, then demonstrate why it did not work for TLQ. We ask our database to find each file matching the name of the shared filesystem (`/disk/`) which every failed task accessed. This is done with two table joins in order to match tasks to files (a many-to-many relationship). Relating this to TLQ's data model (which represents each

component as a queryable node in a property graph), the properties are explicitly identified by the user in the query in `SELECT` and `USING` statements. The links are made apparent on `JOIN` statements. We receive a set of failed accesses which are presented as column-delineated rows. We present a condensed representation of the output received.

Which failed tasks accessed the `/disk/` shared filesystem?

```
SELECT * FROM tasks LEFT JOIN tasksToFiles USING (taskid) \
LEFT JOIN files USING (fileid) WHERE tasks.failures = 1 \
AND files.name LIKE '%/disk/%';


taskid|...|fileid|procid|masterid|  name  |
  72  |...|  307 |  315 |    1   |'/disk/'|
  16  |...|  307 |   35 |    1   |'/disk/'|
...
```

While SQL is known to be incredibly performant [7, 122] even in distributed applications, we found it was insufficient for TLQ as is. Knowing the schema of any possible log type is impractical to realize. TLQ sidesteps this by having the parsers *de facto* handle component log schema (which may change over time). Each parsed metalog contains key metrics at the top level rather than following a static schema. Further, the query presented demonstrates a glaring incompatibility between TLQ and SQL: `SELECT *` is impractical to implement. There are two methods by which we can approximate the intended use of such a powerful statement. First, we can interpret `SELECT *` to mean, "Query the sources I have been told exist." This is essentially how TLQ's user client operates. We can only query log servers the client knows exist. However, this fundamentally changes SQLite's operation (and radically changes the semantics of queries). We decided it would be preferable to

find a querying engine which more closely matched TLQ's needs than attempt to engineer SQL in ways it was not designed to handle.

The second method would be to approximate constraining the open system to a closed system. Log servers which contained logs the client knows exist would be periodically queried to send a copy of each log file (and the parsed metalog) to the user client. SQLite would then perform queries upon this local, closed system rather than out to the open system. However, this negates the key benefit of TLQ which is to leave all debug logs in place, querying them only when needed. This idea was a nonstarter given it went against the design of TLQ.

7.2 RethinkDB

RethinkDB's ReQL querying language is based on JavaScript and has a functional approach to querying data. It is worth noting it is more verbose than the SQL query used in SQLite to find the same information. The general approach is the same, however. Two table joins were performed to find the files which each failed task accessed. In contrast to the SQL `JOIN ... USING` syntax, ReQL allows for a user to specify an anonymous function which resolves the join clause. Similar to SQLite's approach in realizing TLQ's data model, the properties are explicitly defined in the user's query within the `filter` calls and anonymous functions of the joins. The links are made readily apparent in the `innerJoin` calls. This allows for more flexibility at the cost of verbosity. We show the query asked of the RethinkDB instance.

Which files did each failed task access?

```
r.table('tasks').filter({failures: '1'})
.innerJoin(r.table('tasksToFiles'),
function (task, file) {
 return task('taskid').eq(file('taskid')) })
.innerJoin(r.table('files'),
function (taskFile, file) {
 return taskFile('right')('fileid')
 .eq(file('fileid')) }).zip()


[{ fileid: '307', id: 'dc70f638-...',
 left: { category:'default', name:'/disk/',
  command:'ltrace-wrapper ./fscheck', ...
 right: { fileid:'307', id:'067ba70a-...',
   procid:'330', taskid:'77' } },...
```

RethinkDB comes a step closer than SQLite in that it is schema-free. Its output is also JSON, which matches the format of the metalogs stored at each log server. Further, its JavaScript querying interface makes the querying experience extensible. We can seemingly add the necessary functionality into ReQL as JavaScript functions. From first impressions, this seems like a good fit for TLQ.

However, as with SQLite, we found RethinkDB insufficient. The primary reason is the same as one from our experience with SQLite: it cannot effectively be applied to an open system. Even though we would not have had to override a key piece of the querying language (unlike with `SELECT *` in SQL), our options remain the same. Either we would have to approximate an open system by collecting all the debug logs locally or modify ReQL to resolve HTTP requests to known log servers. Modifying ReQL would involve changing backend code.

Rather than querying a specific database instance (the `r` variable in the ReQL query contains a reference to the database), ReQL would need to resolve individual HTTP queries over an array of known log servers. This would change the foundational paradigm of RethinkDB which is to centrally collect data and add it to its local database instance. For this reason, the approach to create a local, closed system from the current state of the open system would be the same as in SQLite and falls short of TLQ's needs for the same reason.

7.3 GraphQL

Unlike the previous two technologies, GraphQL is a property graph traversal engine. It is designed to retrieve requested properties from graph nodes and, when needed, follow the links at those nodes to other related nodes, performing the same actions (repeating this process until either exhausting all outgoing links or until a certain depth is achieved). GraphQL's query structure is the most distinct from SQLite and RethinkDB however it is about as verbose as the SQL query. There are no tables in its implementation, only a raw JSON document.

A query in GraphQL is also expressed as JSON, but it establishes a hierarchy. By this we mean the user specifies some entrypoint node (we present the case of a Work Queue master as an entrypoint) and dictates a traversal path in their query. In this example, the master has tasks it submits which in turn run on a worker and use files. The query follows this same order of traversal as a hierarchy, and the result received (also JSON) matches this hierarchical structure. We provide a GraphQL query asking to retrieve each failed task's `taskid`, the `address` of the worker it ran on, and the `name` of the file accessed only if it matches the name of our shared filesystem (`/disk/`). GraphQL's approach to the conceptual data model is different from SQLite and RethinkDB. In GraphQL, the user must explicitly state which properties of each record they would like retrieved. The links are represented

hierarchically rather than through joins. This graph walk approach to querying more closely matches the hierarchy of components in many distributed system applications.

Which failed tasks accessed the `/disk/` shared filesystem?

```
query { master {
 tasks(failures:0,conditional:">") {
  taskid
  worker { address }
  files(name:"/disk/",conditional:"match") {
   name }}}}


result {
 "data":{ "master":{ "tasks":[{ "taskid":1,
 "worker":[{ "address":"localhost:33172" }],
 "files":[{ "name":"/disk/" }] }, ...
```

The hierarchical structure of GraphQL is an especially appealing feature relative to the SQL and NoSQL approaches. The query informs the user of the control flow of its resolution. In short, the form of the input directs the form of the output. We initially settled upon GraphQL as TLQ's querying engine, even combining it with SQLite as a storage layer for performance gains. GraphQL's query structure is both reasonably human-readable *and* makes links between components a core part of the query itself. SQL, however, offers better performance which is why it was used as GraphQL's record storage layer. It has been developed over a longer period and has undergone many iterations of performance optimization for what it does. Table 7.1 succinctly demonstrates the difference in roundtrip query time between GraphQL using SQL as a storage layer and GraphQL using JSON as a storage layer (the default setup for GraphQL). The records queried in this example are stored locally with the query executor.

TABLE 7.1

QUERY ROUNDTRIP TIME FOR SYNTHETIC DATA.

Number of Records	GraphQL + SQLite	GraphQL + JSON
100	0.007s	0.007s
$1,000$	0.018s	0.019s
$10,000$	0.134s	0.602s
$100,000$	1.098s	198.643s

However, there were significant barriers to adopting GraphQL. Primarily, it was unclear where the schema enforcement mechanisms and traversal engine of GraphQL ended and where our user-defined query resolution mechanics began. The framework is incredibly extensible by virtue that all resolution mechanisms are written not by the GraphQL developers but by its users. In this case, we (as users of the GraphQL engine) developed the resolution mechanisms for the iteration of TLQ which used GraphQL. This, in turn, means the efficiency of query resolution is only as good as the user can express. One virtue of this roll-your-own approach is that GraphQL can be applied to an open system. The TLQ client is informed of which log servers exist. This can be passed to GraphQL as a list of hosts to query, and since we are writing the resolution mechanics it is straightforward to include HTTP requests as part of each query evaluation. However, care must be taken here. GraphQL is meant to be asynchronous, so these requests must be non-blocking.

Whereas SQLite and RethinkDB provided too much structure to the point of constricting TLQ's effectiveness, GraphQL did the opposite. It did not provide enough constraint to the point it became unclear how to measure the performance of GraphQL. Figuring out how much of the performance impact was due to our res-

olution mechanics and how much was due to GraphQL's internal code became an obtuse question which could not readily be answered. Due to this, GraphQL was not adopted as TLQ's query language.

7.4 JX

This leaves JX (JSON eXtended)[1] [114]. It adds Python-style list comprehensions, variable resolution, and handy functions which are all evaluated by the JX parser into a resulting JSON document. JX also allows variable bindings to be introduced via *contexts*, or objects consisting of key-value pairs, where the value is bound to a variable with its name given by the key. Additionally, useful built-in functions (such as string formatting) allow for JX expressions to become incredibly rich in expressive power without being incomprehensible to a reader.

As JX is not a general-purpose programming language, it does not allow unbounded iteration, recursion, and other control flow structures. Rather, JX serves as a compact representation of complicated and deeply-nested data structures. Since a JX document has the same general form as the JSON structure it produces, it is straightforward to incrementally build complicated expressions by generalizing a JSON structure. We provide brief examples of JX functions applied to atomic values, array, and objects. In the final example the object argument is first evaluated, then the variables `x`, `y`, and `z` are bound for the duration of the call to `eval()`.

---

[1]Documentation for JX can be found at: `https://cctools.readthedocs.io/en/latest/jx/` or at `https://ccl.cse.nd.edu`.

```
1 + 1 == 3;
  => false
range(10);
  => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
len([1, 2, 3]);
  => 3
[x*x for x in range(5)];
  => [0, 1, 4, 9, 16]
[{"pet": x, "count": len(x)} for x in ["dog", "cat", "bird"]];
  => [{"pet": "dog", "count": 3}, {"pet": "cat", "count": 3},
       {"pet": "bird", "count": 4}]
{"type": worker, "logs": [template("/tmp/task{x}.log") for x in
  range(5, 8)]};
  => {"type": "worker", "logs": ["/tmp/task5.log",
       "/tmp/task6.log", "/tmp/task7.log"]}
eval(y == 16, {"x": 8, "y": x * 2, "z": "test_value");
  => true
```

Before adding the querying functionality necessary for TLQ, JX was used as an intermediate representation for Makeflow scientific workflows. The condensed JX representation would be expanded into a full JSON representation of a workflow which could then be executed directly by Makeflow. The expressions, list comprehensions, and built-in functions provided by JX made it possible to compactly represent quite lengthy workflows. To more effectively make use of the parsed JSON metalogs at each log server, we implemented querying capabilities for the JX language by way of adding new built-in functions to the language.

Three basic JX functions opened up the possibility to perform distributed queries

upon linked data: `fetch`, `select`, and `project`. The `fetch` function is given a path or URL as an argument, retrieves the document located at that path/URL, and returns a parsed JX object of that document. The retrieved document at the URL or local path must be valid JX or JSON to be successfully returned to the user. The `fetch` function is necessary for retrieving documents across the open system as well as retrieving documents from the links of an object currently being queried.

The `select` function takes a Boolean expression (of arbitrary length and number of predicates) and a list of objects upon which the expression is evaluated. Each object is used as a context for evaluating the Boolean expression, and only those objects for which the expression evaluates as `true` are included in the returned list. `select` thus serves to filter the list of objects based on a given predicate. The list of objects is often resolved from a call (or calls) to `fetch`. A `select` call on a single object must still be treated as a list of objects (simply of size one).

The `project` function takes an arbitrary JX expression and a list of objects. The expression is evaluated using each context object in turn, where the expression can use any of the fields from the context objects. This expression can be used to extract values from the context objects, or perform more complicated transformations. The resulting, evaluated JX objects are accumulated in a list of the same length as the passed in list of contexts. As with `select`, `project` may operate on just a single object so long as it is still encapsulated within a list. We provide some brief examples of these functions in practice.

```
[fetch("doc.1.json"), fetch("doc.2.json"),
  fetch("http://doc.3.json")];
  => [{"type": "ltrace", "failures": 2, "top_error": "SIGSEGV"}
     {"type": "worker", "failures": 0, "top_error": ""},
     {"type": "ltrace", "failures": 74, "top_error": "ENOSPC"}]
select(failures > 0,
  [fetch("doc.1.json"), fetch("doc.2.json"),
    fetch("http://doc.3.json")]);
  => [{"type": "ltrace", "failures": 2, "top_error": "SIGSEGV"},
     {"type": "ltrace", "failures": 728, "top_error": "ENOSPC"}]
select(failures == 0 and type == "ltrace",
  [fetch("doc.1.json"), fetch("doc.2.json"),
    fetch("http://doc.3.json")]);
  => [ ]
project(failures,
  [fetch("doc.1.json"), fetch("doc.2.json"),
    fetch("http://doc.3.json")]);
  => [2, 0, 74]
project({"type": type, "valx2": value * 2},
  [fetch("doc.1.json"), fetch("doc.2.json"),
    fetch("http://doc.3.json")]);
  => [{"type": "ltrace", "valx2": 20}, {"type": "worker",
       "valx2": 0}, {"type": "ltrace", "valx2": 148}]
```

Beyond `fetch`, `select`, and `project`, we also implemented two additional helper functions for JX's querying capabilities: `schema` and `like`. The `schema` function takes as input a JX object and returns an object which contains a key-value pair of

each key of the input object and its data type (e.g. "exit_status": "integer"). The `like` function is used to perform a regular expression match to a string similar to SQL's `LIKE`. It returns `true` if a match was found, `false` otherwise. We provide brief examples of `schema` and `like`:

```
schema(fetch("doc.1.json"))
  => [{"type": "string", "failures": "integer",
       "top_error": "string"}]
like("potato", "tomato")
  => false
like("*SIG*", project(top_error, [fetch("doc.1.json)])[0])
  => true
```

This simple, transparent parsing and expression evaluation provided a lower barrier to entry than modifying the other technologies used previously. The querying functions implemented in JX were, for the most part, purely additive. The existing evaluation mechanics did not have to be modified, rather new cases were added to them. New functions were added to the host of JX's built-in helper mechanisms rather than modifying existing ones to fit TLQ's use case. Fetching data via HTTP was included to allow JX queries to pull data from machines in an open system. Selecting and projecting functions were included to perform data transformations inspired by the foundational theory behind SQL. With these three functions, JX was extended to perform distributed queries.

Adding the required functionality to JX was more straightforward than it would have been for SQLite and RethinkDB. Instead of having to radically alter the backend of an established technology, we could simply plug in our newly created mechanisms to the existing JX parser so they would be recognized when a user submitted a query to the TLQ client. However, unlike GraphQL, the evaluation mechanism was

already in place and well understood. It can be effectively measured, and its results are predictable. The added querying functionality interacted with this transparent evaluation mechanism whereas in GraphQL the evaluation was entirely user-defined, and capturing the resolution in the GraphQL backend was not feasible.

With three critical functions (`fetch`, `select`, and `project`), JX provides the necessary mechanisms to retrieve relevant debug log data which live in an open system, perform transformations upon them, and output data to the user in a digestible manner. The additional functions `schema` and `like` provide quality-of-life improvements which make JX more convenient to use as a querying language. We demonstrate JX in action in Chapter 9. The result of implementing querying for TLQ in JX is a simple yet powerful querying mechanism for open distributed systems.

CHAPTER 8

TLQ'S WEB INSPIRED APPROACH

Implementing log discovery and log custody were critical for enabling the use of existing tools in an open distributed system. With log discovery enabled, we are able to figure out *where* debug logs exist. Log custody provides a way for ephemeral components' debug output to exist after that component's lifetime. These combined allow us to use TLQ to run troubleshooting tools on these logs spread out across the open system under study. However, they are only parts of the whole functionality TLQ provides.

8.1 Problem Introduction

In Chapter 6, we noted in some key lessons learned that it was not possible to truly take advantage of the relationships between components with only log discovery and custody. They provide mechanisms for inspecting and retrieving debug output, but they do not give us the full breadth of exploration which would make troubleshooting in an open system convenient. It would be preferable if we could follow the relationships (i.e. the links) between components which interacted in the system. To this point, that information is contained within the JSON metalogs for each component, but TLQ has no mechanism to take advantage of these.

We implemented JX at the user client in TLQ to address this previously missed opportunity. In addition, we iterated over the log discovery and custody mechanisms, making them more effective. We also made each parser more full-featured. Each one produces not only top-level metadata, but they also convert each line of the debug

log being parsed into a JSON representation (addressing the uniform log format issue in Chapter 6). Further, we modified Makeflow and Work Queue to explicitly record their own UUID in their debug log *and* record the UUIDs of components they interact with at runtime. This again addresses a key lesson learned from Chapter 6. The implemented JX querying capabilities are inspired by the structure of the World Wide Web (the most large-scale contemporary open distributed system humanity has created). Indeed, the architecture of the web can be mapped to TLQ's architecture quite readily, demonstrating how key fundamental aspects of the web can be applied to TLQ's use case.

Through TLQ, we provide mechanisms for log discovery (how to find debug logs), log custody (how to access those logs), and a rich querying experience for linked data (one log references another) akin to the World Wide Web. To reiterate: log servers advertise the existence and location of components and their debug output directly to the user, giving them a unique URL for each log created by the system. A log server persists on each machine in the cluster, cloud, or grid, dedicated to monitoring components it is told about. In addition, they take custody of the logs created by components. If the server has the proper permissions, it will take direct possession of the log. The given path in the component's command line string to is replaced with the unique file name (i.e. `[UUID].log`) in the server's working directory. If it does not, the server periodically copies over updates to a version of the log in its working directory.

Logs tracked by TLQ are periodically parsed, and the results are stored as JSON. These JSON documents can then be queried using the JX language, providing a single interface for all logs. The querying possible with JX allows TLQ to provide a web approach to open systems troubleshooting. By this we mean JX queries can traverse links within one document to other documents, even if they are located on different machines. As the query is resolved, it will fetch and evaluate along relevant

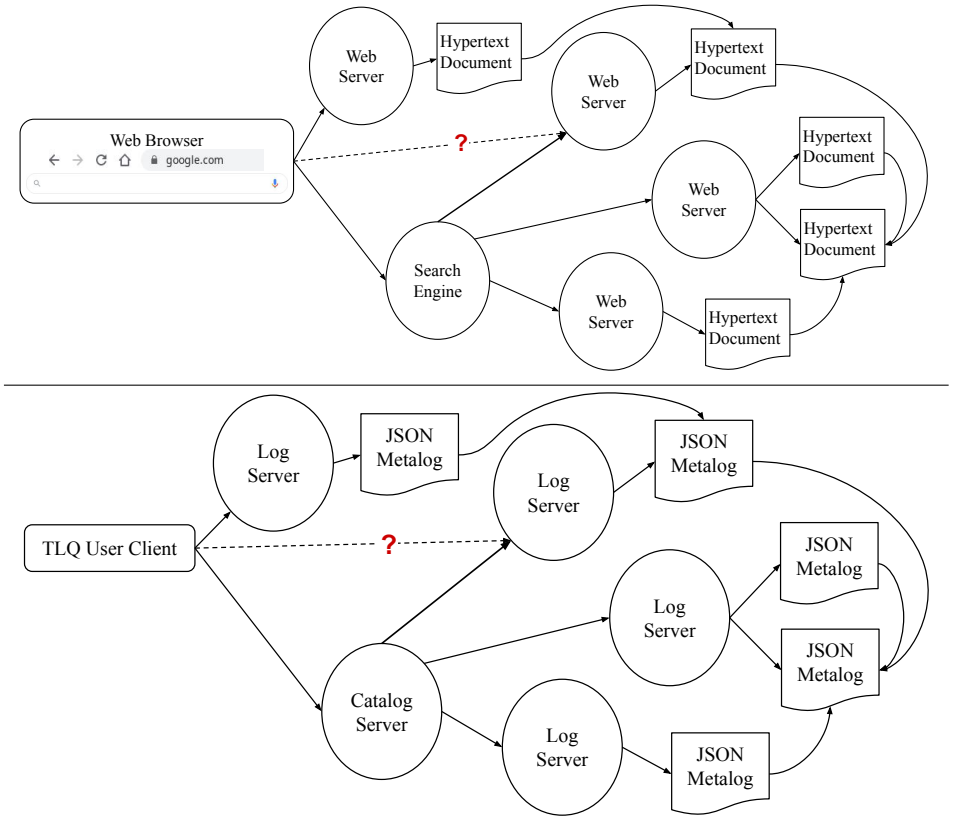

Figure 8.1. Web architecture and TLQ querying architecture. Finding a web document: directly access its URL or ask somebody who knows how to find it (like a search engine). Finding a TLQ JSON metalog: directly access its URL or ask a catalog server which knows about active log servers. Both documents and metalogs may contain links to others.

links (i.e. perform a graph walk), until a final list of JSON objects is returned as the fully evaluated result. This is accomplished by TLQ's assigning of unique URLs to every log advertised to log servers.

8.2 A Web Inspired Approach

The web is, at its core, an open distributed system. It is a massive, decentralized network of components (sites, services, etc.) and HTML documents addressable by unique names (URLs). So long as we know a component's name, or can find its name,

we are able to traverse the web. TLQ is designed to explore open distributed systems the same way. Debug logs are given unique names, and a user can request those logs or query them by name to troubleshoot their system.

The top diagram of Figure 8.1 shows a simplified architecture of the World Wide Web. Using a client (e.g. a browser) to access a hypertext document on the web requires that we either know the name for that document (its URL) or have somebody we can ask who may know its name (like a search engine). Once we have the URL, we then access the document via its home server. Within a hypertext document are links to other documents (perhaps hosted on *other* servers across the world). Our browser can be redirected to these other documents because we know their names from the links (their URLs).

We follow a similar approach with TLQ. Each log is parsed at its local log server into a JSON metalog which contains metadata about the component which the log represents as well as JSONified representations of each line in the log. The JX language can then be used to query these metalogs. The bottom diagram of Figure 8.1 shows how TLQ emulates the architecture of the World Wide Web. JSON metalogs are linked data and can be queried individually or iteratively by following the links contained within a document. A user can directly access a metalog via its URL, or it can ask a catalog server for a list of active log servers. As log servers run, they periodically push a heartbeat to a publicly known catalog server to advertise their existence. The client can ask individual log servers for a list of the logs they are watching, returned as a list of URLs. Once a metalog is retrieved (via the `fetch` JX function), its links can be followed by subsequent calls to `fetch`.

8.3 Log Record Data Model

Each TLQ log server periodically parses the logs it watches in order to produce records which are stored locally as JSON metalogs. Each record follows a common

data model, no matter what type of component the record represents (such as a data replica node, a traced computational process, or a workflow manager). Each record has a type, a unique ID (in our case a URL), a set of properties (stored as key-value pairs or lists of key-value pairs), and a set of links (a set of two key-value pairs each).

The record's type corresponds to the kind of component the record represents. The URL is generated by the monitor script from a UUID and its own address, after which it informs its local log server a log with a given UUID and URL now exists. This UUID is used as a unique identifier to establish links to other records. The properties are key-value pairs which can correspond to state information about the component such as total runtime, exit code, resources consumed, and component-specific debug metrics. Links represent the relationships a record has to *other* records. They are composed of the linked record's type as well as its URL. Properties and links cannot be removed once detected since they may be relevant for troubleshooting. We present two examples of TLQ's data model format: a batch job and a file.

```
record {
 type: batch_job, url: "http://...",
 jobID: 8672,
 command: "run.exe",
 ...,
 links: [ {type: batch_system, url: "http://..."},
  {type: submitter, url: "http://..."},
  {type: process, url: "http://..."}]
}
```

```
record {
 type: file, url: "http://server/log.json",
 path: "/path/to/file.txt",
 size: 10424, lastAccess: 177283,
 ...,
 links: [{type: directory,url: "http://..."},
  {type: user, url: "http://..."},
  {type: user, url: "http://..."}]
}
```

TLQ's data model is similar to (RDF) [87]. Links in TLQ JSON metalogs can be thought of as relationships between RDF resources. However, there is no concept of a triplestore (which is the typical data structure for storing RDF records). All properties and links for a record are contained directly within that record rather than being separate, linked records as in triplestores.

This data model is also similar to JSON-LD [73, 121]. JSON-LD has document types called *contexts* which are like TLQ's component types. These map to pre-defined schema linked by URL to a document in JSON-LD whereas TLQ's parsers essentially create the schema of the component types. IDs in JSON-LD are URLs like in TLQ as well. However, the context schema applied to TLQ would be redundant information since we also need parsers at each log server. These parsers define the schema at runtime, and they may change over time to address alterations to a log's format (whereas JSON-LD context documents as proper schema should not). We chose to use JX without pre-defined schema (as in JSON-LD) as it was a more straightforward approach which mapped better to the ever-changing nature of open systems. Other frameworks and data models (though they may map well to the problem, such as JSON-LD) were not as straightforward to implement.

8.3.1 Query Model

The main approach of TLQ is to keep all logs in place at the point of creation, performing queries when possible. A query in TLQ is either a distributed invocation of a command line tool on a log (i.e. a traditional troubleshooting utility) or a JX query on a JSON metalog (or set of metalogs). Rather than trying to transform an open distributed system to a closed one (an incredibly difficult problem with diminishing returns), we focused on leveraging basic information necessary to have machines in an open system communicate: names and a method to contact each other. Once a component has a unique name (a UUID) and a means of accessing it (a URL), a user can then submit queries directly to the URL which is serviced by a log server. This means we treat troubleshooting open systems as a set of distributed queries. The TLQ approach enables a user to keep the tools they like so long as they can be invoked remotely on the logs.

8.4 Implementation

As was introduced in Chapter 6, the TLQ architecture is composed of four parts: a log server, a set of log parsers, a user querying client, and a monitor script which advertises debug output to a log server and reports back to the client where that log ended up. Refer back to Figure 6.1 to see how these parts interoperate. The general structure of these parts remain the same, however we add new querying capabilities at the user client.

The user client provides the capability to use traditional troubleshooting tools of a user's choice upon logs (and metalogs) stored across different log servers. We demonstrated how `grep` can be used to troubleshoot an open distributed system [65]. Before being able to use a troubleshooting tool, the client must have available logs advertised to it. This list of logs, represented by unique URLs, is accumulated by

a minimal HTTP server running within the client which accepts messages from the monitor script advertising a log has been accepted by a log server. In addition to sending queries out to log servers, the client can also execute JX expressions. Using the functions defined previously (`fetch`, `select`, and `project`), a user can chain together powerful JX expressions to extract highly specific and interesting data from the JSON metalogs across different log servers. We demonstrate this directly on two different systems under study in Chapter 9.

The monitor script lands on a machine by a work scheduler and is in charge of advertising a log's existence and executing a given command. The monitor is given a home address and port (the location of the user client), the working directory of the log server (this location is known *a priori* by the user or administrator who initialized the log server), the name and type of each log created by the command, and finally the command to be executed. It then attempts to replace any occurrence of the given log names in the command string with a generated UUID for that log. The log server is informed by the monitor via HTTP the log names, log types, command string, UUID, and whether the log was able to be replaced in the command string. The monitor terminates before the command is executed if no log server is found. Otherwise, the command is then invoked, starting up a component of the system.

We used the Makeflow workflow management system and the Work Queue master-worker framework in our evaluation of TLQ. Makeflow and Work Queue interoperate similarly to the layout shown in the initial notional example in Figure 1.1. While each component (the workflow management system, the master process, the workers, and individual tasks) interacted to varying degrees, these interactions were not explicitly captured in the debug output of these components. This led to it being impossible to concretely establish a link between two components. Makeflow, the Work Queue master, and the Work Queue worker code were altered to look up the local TLQ log server and retrieve the URL for their respective logs. The first time one of these

components interacts with another, they advertise their TLQ URL and request the other's URL in return. These URLs are then recorded in the respective components' debug logs, thus explicitly linking two components together.

8.4.1 Server Requests

A log server can receive a number of requests from clients. It handles each request over HTTP. The most basic of these is to send back raw logs and JSON metalogs to the client. For this functionality, TLQ provides a means to transparently advertise the location of all logs in an open system and allow for their retrieval. This alone is useful for open systems.

In addition, a user can submit arbitrary command line queries to a log server. A command line query is an invocation of a command line tool. This allows for a user to bring along their tool(s) of choice rather than being locked into a domain-specific language for TLQ. For example, `grep` can be called upon logs across log servers [65].

JX query requests are designed to operate specifically on JSON metalogs. JX queries are evaluated at the client, however some JX functions may interact with log servers to retrieve data. The `fetch` function, for example, directly accesses a log server to pull JSON documents to the client, which the log server provides for on-demand access.

TLQ log servers also have a user-centric data retention policy. A user can send a request to a log server to have it delete a log. Upon receiving this request, the server will attempt to remove both the raw log and JSON metalog. Unless overridden by an administrator, log servers will hold all logs unless told otherwise by a client. This also extends between invocations of the log server. If a new server is configured to use the same working directory as a defunct log server, it will read through the log deposits manifest and take ownership of any logs in that directory.

8.5 Lessons Learned About Log Design

Having fully described the architecture of TLQ, we return to one of the lessons learned in Chapter 6: creating specialized parsers for each type of debug log was a tedious process. The metadata a parser pulls out of a log is informed by some domain knowledge. The person writing the parser must understand the format of the log, the semantics of relevant lines, and which pieces of metadata are important *and* able to be found or derived. This is not always straightforward, especially considering how diverse different log formats can be.

One change to debug logs which would make parsers less tedious to write would be establishing a transactional format. By this, we mean each line of the debug log contains a complete record of a particular event. This may include data such as a timestamp, a human-readable status report, all components involved in the event, etc. Since this is not enforced by TLQ, a log format which contains the relevant context for an event across *multiple* lines of debug output would require domain knowledge by the parser writer to put all those pieces together. Instead, a transactional approach would allow the parser writer to focus on which events (and by extension particular, *individual* lines) are relevant for metadata reporting, lowering the barrier to entry for users writing new parsers.

Related to adopting transactional logs is establishing a more universal log format. There exist many popular formats for debug logs already such as JSON and XML. TLQ parses each log into JSON to establish this universality. JX can directly query, so making all debug logs JSON directly benefits the user when troubleshooting. If each debug log were formatted in JSON at runtime, there would be no need for specialized parsers. Instead, the raw logs would be directly queryable.

The logs produced my Makeflow and Work Queue used by TLQ approach these two benefits. Each line contains a timestamp, the source of the event being logged, and a human-readable message relating to that event. They are nearly transactional,

however some events such as tracking changes in a Work Queue task's status, sometimes stretch across multiple lines of the log. Additionally, their nearly transactional nature makes them very straightforward to transform into JSON. Altering both Makeflow and Work Queue to output JSON debug logs rather than their current format would be fairly approachable since most events' contexts are currently formatted for one line of debug output. Changing *all* types of logs involved in a system to JSON would involve more work, akin to implementing the parsers, and would require the requisite domain knowledge.

A final benefit to log design would be making links between components explicit. If a log does not provide explicit reporting of a component interaction, it may be missed by the parser. Or worse, there may not be enough information to make that connection, meaning that information is irretrievable. We altered Makeflow and Work Queue processes to report their TLQ URLs when communicating with each other. This makes the linking between each component easy to recognize for parsers (or for direct querying if the log is already JSON) since the type of the interacting component *and* how to query it are directly provided by the log. No additional investigation needs to be done by the user to find related components.

8.6 Conclusions on the Design of TLQ

We presented TLQ (Troubleshooting via Log Query), an architecture for troubleshooting open distributed systems. This is made possible through a web approach which allows queries to follow links from one debug log to another, which may span across the machines used by the system. We demonstrated through notional example why troubleshooting distributed systems is so difficult in Chapter 1 and have now shown how the TLQ architecture addresses the inherent complexities of them.

With TLQ we have demonstrated the capability to effectively troubleshoot *open* distributed systems where the membership of machines is not constant, the placement

of work is unknown *a priori*, and resources from multiple independent jurisdictions may be utilized. By using a web approach where logs are addressable via URL and have links within them to other logs (potentially on other machines), we are able to traverse all logs in a system. This is, at best, impractical with a centralized approach. At worst, it is impossible. TLQ makes this not only possible, but feasible within the complexities inherent to open distributed systems.

CHAPTER 9

CASE STUDIES OF TLQ

Having defined the problem space for open distributed systems troubleshooting, the corpus of related work, a concrete example of a distributed system problem in action, the performance of key mechanisms needed to resolve the problem space, and the facets of TLQ's architecture, we conclude with two case studies of TLQ in action providing log discovery, log custody, and a web approach to querying related components of a system. The first case study presents a common user headache: uncovering misbehaviors when scaling an application from serial to distributed operation. The second case study demonstrates the usage of JX in depth to troubleshoot the previously uncovered issue in the Lifemapper biodiversity workflow.

9.1 POV-Ray

Scaling an application from a serial program to a distributed system is often a time-intensive activity. Unforeseen behaviors and errors can crop up when transforming a task typically ran in serial to one which runs in parallel. Further, we typically begin interacting with machines we have no means to physically inspect once we switch from a serial mode of operation to a distributed system. We demonstrate how TLQ can be helpful in making transparent the misbehaviors that makes themselves known due to scaling up an application, allowing the user to more thoroughly investigate any issues.

POV-Ray is a ray-tracing program which takes a description of a scene (via a domain-specific language) and performs a realistic rendering of that scene. A scene

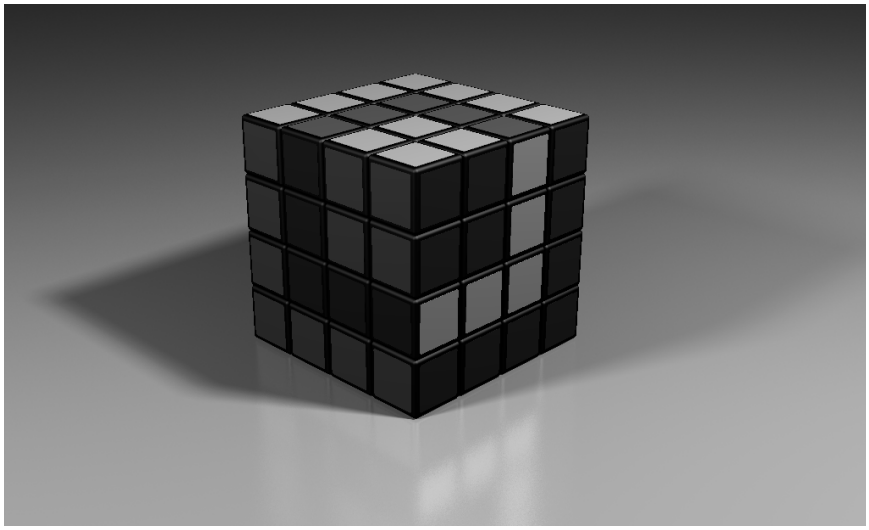

Figure 9.1. Rendered POV-Ray frame.

can be composed of multiple frames to create a video, and the markup language is used to give POV-Ray directions how to manipulate the scene between frames. Figure 9.1 shows a single rendered frame of a video of a Rubik's Cube performing rotations (as if attempting to solve itself).

When executed serially, each frame is rendered in order one-by-one. However, each frame can be rendered independently; POV-Ray has a description of each frame in the scene via the scene description. We can also render an arbitrarily long video by having frames be repeated, but this requires duplicate renderings (added work). POV-Ray is an embarassingly parallel (also called pleasingly parallel) application. There are no dependencies between each frame render which means it should closely follow the Gustafson-Barsis Law [38] as discussed in Chapter 3. It should experience greater and greater speedup as more computing resources are provided to POV-Ray.

The ideal scenario would be to scale up a serial implementation of a ray-tracing pipeline to a massively parallel pipeline which can run *all* frame renders at once. Depending on the scale, this may overwhelm the machines in the cluster, cloud, or grid. So, as a caveat, we should aim to scale up POV-Ray to as large a scale as the

underlying distributed system can handle, submitting a reasonable number of frame renders in parallel. We demonstrate how TLQ can assist in troubleshooting issues that come up as a result of scaling up a previously serial application to a distributed system.

Our initial serial configuration for a POV-Ray video rendering pipeline was quite simple. A script was written to take a few key input options: the input scene definition, the video to be output, the number of frames, the height of each frame, and the width of each frame. POV-Ray was then called to render each frame in sequence. This was the most time consuming portion of the work. Finally, a video stitching all frames together was rendered using the `ffmpeg` program.

Each frame took approximately 35 seconds to render via CPU. Rendering a minute long video took 600 frames. Rendering a more interesting five minute video took $3,000$ frames. It quickly becomes obvious running this application in series is an incredibly inefficient use of time. It would be preferable to run the rendering process in parallel until we are ready to combine all the frames into a video.

9.2 Scaling Up to Parallel Work

To scale up this pipeline, we used the HTCondor batch job system. A job in HTCondor is like a Makeflow rule or Work Queue task. It is a definition of work to be done. A job definition includes required inputs, expected outputs, and the command to run. In addition, a job includes additional metadata such as resource specifications and requirements for the machine which will eventually run the job. This additional information ensures only machines with adequate hardware and environment configurations will execute the job.

These job definitions are communicated to a central manager which plays the role of matchmaker. It will determine which machines are able to complete the work described by the job and inform the submitter how to connect to one such

machine. The submitter and the executor (both daemon processes running on their respective machines) spawn a connection handling process to begin communication with each other. Once this connection is established, the job definition and inputs are transferred to the machine running the job, the job is executed, and outputs are pushed back to the submitter. Figure 9.2 demonstrates this architecture.

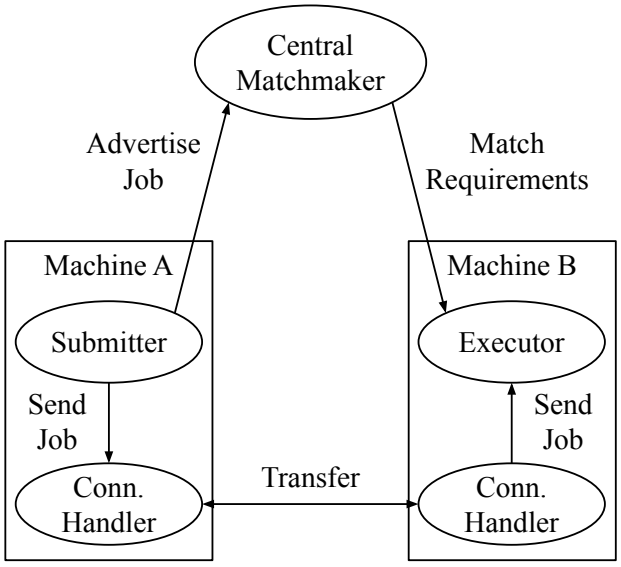

Figure 9.2. HTCondor architecture. Submitter advertises jobs to the matchmaker. Matchmaker matches job requirements to available machines. Both submitter and executor spin off connection handlers for transfers.

Many jobs can be submitted at once, allowing a scale up the parallelism in the rendering pipeline to as large as HTCondor will allow (since POV-Ray is a pleasingly parallel application). In the serial script, we give each frame its own HTCondor job definition. Instead of executing each frame render locally, we submit its job definition, letting HTCondor handle it from there. This submission is non-blocking, allowing submission of *all* frame renders without having to explicitly wait on any results to come back. Once all jobs are submitted, we wait for all jobs to complete (i.e. all

frames rendered). Once all jobs are complete, `ffmpeg` is used to create the video locally.

An HTCondor job is an example of an ephemeral component. Once the job is completed, its working directory is cleaned up (including its debug log unless explicitly defined as an expected output). TLQ can help keep track of all these ephemeral components. It can be set up to help troubleshoot issues that crop up with this new, scaled up job submission script. For each job definition, we can specify a certain set of required machines. HTCondor's matchmaker will only connect the jobs to those machines (or will have the jobs wait until one of those machines is available). We can set up TLQ's log servers on each of those specified machines, ensuring each job's debug log will be captured.

We initially used 10 TLQ log servers (thus 10 machines) when testing the scale up. Even at this seemingly small scale, we uncovered an issue thanks to TLQ. POV-Ray had an incomplete specification of its jobs. There was an implicit input file (`WRC-RubiksCube.inc`) which was uncovered within the debug logs of each attempted job. This is an example of a misconfiguration failure. The scene description includes this input file, but HTCondor and the submit script were never told about it. So, when the jobs began to execute remotely, POV-Ray failed almost immediately on each job. It could not provide even a partial result without that input file. We show the debug log lines which correspond to this implicit input file.

POV-Ray Debug Log's Helpful Output

```
Possible Parse Error: Cannot find file 'WRC-RubiksCube.inc'.
File: rubiks1920.pov  Line: 54


#include
Parse Error: Cannot open include file WRC-RubiksCube.inc.
```

At the scale of this application (3,000 jobs), it would have been overkill to bring all 3,000 debug logs back to the submit node. Instead, with TLQ, we kept all logs in place and retrieved only a single one for analysis. Because POV-Ray's debug log is meant to be human-readable, a brief inspection of the log was all that was necessary to uncover the root cause of the failure: this implicit input file needs to be explicitly stated in the job description for HTCondor.

9.3 Incorporating Persistent Resources

One aspect of HTCondor which can be improved is persistence of resources. Each job submitted to HTCondor is provided resources once the job lands on the executing machine. After the job is complete, those resources are then released back to the executing machine. This would not be a problem if we were the only ones wanting access to this pool of resources. However, HTCondor's resource pool is somewhat competitive. It is first come, first served. On top of this, each user is given a priority to ensure fairness. If you have been using a large quantity of resources in the recent past, the penalty to your priority will be more significant. After all, the HTCondor pool is for everyone.

Submitting a large enough quantity of jobs can cause HTCondor to slow down how quickly jobs are placed on machines due to priority being affected. Essentially, rendering a large enough number of frames in a short period of time will cause a wait in a queue before successfully submitting more. To mitigate this, we can use Work Queue as a way to hold on to a set of resources and reuse them until we are ready to release them. Each Work Queue worker process we submit is much more long-lived than the ephemeral POV-Ray jobs, and they are able to run the render jobs in the same way as HTCondor. In essence, Work Queue creates a private cluster of resources where each worker is submitted as a job to HTCondor. Once these workers land on machines, they hold the resources for the POV-Ray frame renders.

Another benefit of using Work Queue is that we can more accurately track what all is happening when a job runs, its resources used, etc. These metrics can be retrieved from worker logs via TLQ's JX querying mechanism and TLQ's ability to use command line tools remotely. If we were not using TLQ, we would have to wait until each worker ends before being given its debug log (assuming we had even specified that we wanted that debug log back in the HTCondor job definition). This also assumes we knew exactly where the workers landed and where their respective working directories were located. Without TLQ, we can only solve the first problem by asking the Work Queue master process.

However, with TLQ we can check in on a log at runtime. This is particularly useful for somewhat long-lived components (like Work Queue workers) to extremely long-lived components (like system administration tools which are *always* on so long as the machine has power). We demonstrate three interesting queries which help gauge the status of the set of workers *as they run* which before TLQ was an inconvenient task to accomplish at best and impossible at worst.

The first query helps answer a foundational question when operating in an open system: where is everything located? In this case, we care primarily about the placement of Work Queue workers. We can also make this query a bit more helpful by also checking how many failures each worker had experienced so far (since the system was still rendering frames at query time). The second query demonstrates a very simple metadata operation which can be retrieved after parsing a log: a component's uptime. The final query uses the capability of TLQ to run command line utilities. From the first query we saw one worker in particular had many failures. A followup using `grep` narrowed down the failures were caused by a lack of storage at the worker. This accounted for all failures at that worker, verified using `wc` to count each individual match captured.

Query 1: Where are my workers, and have they had task failures?

```
project({"host": ip + ':' + port, "failures": failures}, x
  for x in [fetch(y) for y in project(url,
    select(type == "work_queue_worker",
      project(links, project(work_queue_master,
        [fetch("http://log.server.1:11855/jx/db7914e...")])
)[0]))]);
=>
  [{"host": "worker.host.1:9001", "failures": 0},
  {"host": "worker.host.2:9001", "failures": 138},
  {"host": "worker.host.3:9124", "failures": 0}, ...]
```

Query 2: How long (in seconds) has a given worker been active?

```
project(uptime, [fetch("https://log.server.2:11855/jx/...")]);
=>
[3479]
```

Query 3: How many disk storage errors occurred at a given worker?

```
$ query https://worker.host.2:11855/jx/79c6b06f-fb6f-4d29-95...
> grep -E "not enough disk space"|"Failed to put file" \
  79c6b06f-fb6f-4d29-95bc-c4c4e897c1e3.json | wc -l
=>
138
```

9.4 Lifemapper

We revisit the Lifemapper biodiversity scientific workflow as the second case study demonstrating TLQ in action. This execution produced different quantities of data

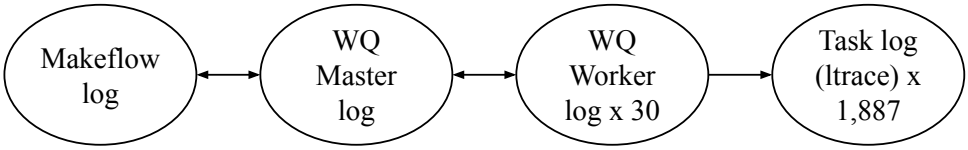

Figure 9.3. Lifemapper link structure. Makeflow and the Work Queue master directly interact, thus know each other's URL. The master and workers advertise their URLs to each other. A task does not know it is being executed by Work Queue. Only the worker can create an outbound link to the task's log. Each log references itself.

(by virtue of running longer) than that introduced in Chapter 6, however the same issue made itself known. Lifemapper was run using the Makeflow workflow management system, which in turn used the Work Queue master-worker framework as its execution engine. Workers were scheduled using the HTCondor batch system while Makeflow and the master process executed at a user-facing head node. Figure 6.3 shows a condensed representation of Lifemapper's structure.

In total, Lifemapper had $1,887$ tasks and had an initial dataset size of 655MB. The tasks themselves consisted of either Java programs or Pythons scripts. Each task was executed by the `ltrace` tracing program, providing a low-level debug log for each task. A total of 30 Work Queue workers were provided to execute Lifemapper tasks, executing across 7 machines. The workflow manager and the Work Queue master executed together on a machine, for a total of 8 in the system. It was executed in 14 minutes and 27 seconds, ending in failure due to multiple Java runtime errors. These errors induced segmentation faults in the child processes of the faulty tasks. Even at a (relatively) small scale such as Lifemapper, a user would have to comb through nearly $2,000$ logs if they wanted to get a sense of what went on in the system, assuming they could even find all the logs themselves.

Figure 9.3 shows the relationships between logs which were tracked using TLQ. Makeflow and the Work Queue Master, although different components, shared the

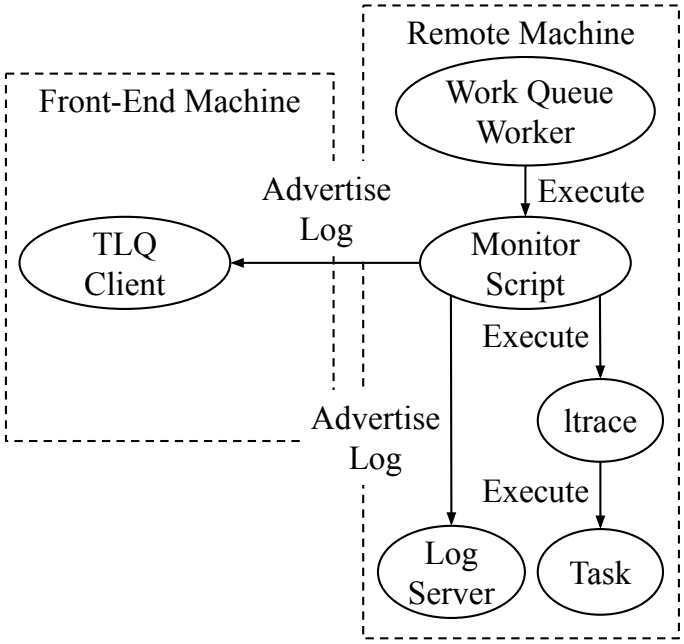

Figure 9.4. TLQ interactions in a Work Queue worker. The worker executes a task from the master. The task is wrapped by the monitor which advertises the task's log(s). The task command is executed via `ltrace`.

same log since they are co-located. They are obviously bi-directionally linked since they know of each other's existence implicitly. Each worker connected to the master and advertised its TLQ URL. The master, in return, sent the workers its URL. This creates a bi-directional link. However, each `ltrace` task did not have awareness that it was executing in a Work Queue worker sandbox. Only the worker knew it was executing a task, and each worker grabbed the URL of each task (from the task's description provided by the monitor script) before forking and executing it. This resulted in a link made from the worker to the task, but the task did not have enough context to make this link bi-directional. Refer back to Figure 1.1 for how tasks are assigned to workers. Figure 9.4 shows how TLQ interacts with a task's execution at a Work Queue worker process.

9.4.1 Interesting Troubleshooting Questions

To demonstrate the querying effectiveness of JX upon the JSON metalogs, we provide examples of interesting troubleshooting questions. The question is transformed into a JX query which produces a fully evaluated JSON result. The results have been truncated for space considerations.

We can use JX to perform regular expression lookups in a particular log. Using the `like` function, we can do a `grep`-like search for alerts of a segfault in a given log. Combined with the built-in `len` function, we end up with a count of all instances of a segfault message in the log, a total of thirteen. Many of the Lifemapper tasks executing Java programs spawn a number of child processes which reported failure due to an array lookup error. This resulted in segmentation faults for each child process at runtime, and the query verifies that.

Query 1: In a log, how many segmentation faults occurred in child processes?

```
len(
  select(
    like(".*SIGSEGV.*", message),
    project(messages,
      [fetch("http://log.server:11855/jx/ffffd2c5-25f2...")])[0]
));
=>
13
```

We can also use `like` to perform a comparison between two tasks' `PATH` environment variables. Misconfigurations of the computing environment are often the cause of failures in distributed systems [104, 139]. When scaling up computations to a new set of computing resources, modifications to the environment are often necessary. A query like this can be used to sanity check that the environment is configured

correctly across the different machines used by the system. The `like` JX function returns a Boolean value, and it compares the value of the `PATH` key within the traced environment variables of two separate tasks (grabbed using two invocations of the `project` function which in turn each call `fetch` on the selected logs).

Query 2: Is PATH the same for these two tasks?

```
like(
  project(environment["PATH"],
    [fetch("http://log.server.1:11855/jx/a0a5bff7-f3bb...")])[0],
  project(environment["PATH"],
    [fetch("http://log.server.2:11855/jx/a217dc04-c8f7...")])[0]
);
=>
true
```

JX queries can not only perform direct queries upon JSON data. They can also be used to traverse *linked* data, enabling a web-like approach to exploring debug output. From the user-facing Makeflow debug log, we can reach each Work Queue worker which ran tasks for Lifemapper. Each link is a JSON object with the keys `type` and `url`. If we only wanted to investigate tasks which ran on workers from a specific machine, we specify the machine name in the `like` function and compare it to the URL of each worker link. After fetching each worker metalog, we can traverse each link of type "ltrace" to retrieve the task traces. Once fetched, we project the list of accessed files for each task.

Query 3: From workers on a given machine, which files were used by tasks?

```
project(files,
    [fetch(x) for log in
      [project(url, select(type == "ltrace", y))
      for y in project(links,
        [fetch(z)
        for z in project(url,
          select(type == "work_queue_worker" and
            like(".*log.server.2.*", url),
              project(links, project(makeflow,
                [fetch("http://log.server.1:11855/jx/a3683...")]
))[0]))])] for x in log]));
=>
  [["/usr/lib64/python2.7/site.so", ...],
  ["/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.265...",
    "/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.265..."],
  ...]
```

We finally demonstrate the richness of JX as a query language with a fairly complex query. We want to determine the commands and URLs of tasks which ran quickly (less than two seconds) across all workers which did work for Lifemapper. This means traversing multiple sets of links as in query 3. The first links, for the workers, come from Makeflow's log. From there, we grab the links to each task (with `type` "ltrace" since each task was executed with the `ltrace` tool). We retrieve each task log, check that its runtime was less than two seconds, and if so return an object with that task's command and URL for further inspection later.

Query 4: For all workers in the workflow, which tasks took less than 2s?

```
project({"command": command, "url": url},
  select(runtime < 2, select(type == "ltrace",
    [fetch(x) for log in
      [project(url,
        select(type == "ltrace", y))
        for y in project(links,
          [fetch(z)
          for z in project(url,
            select(type == "work_queue_worker",
              project(links,
                project(makeflow,
                  [fetch("http://log.server.1:11855/jx/a36...")]
))[0]))])] for x in log])));
=>
  [{"command":"python ./tools/process_points.py points/...",
  "url":"http://log.server.2:11855/jx/4c1adeb8-b85f-45cb..."},
  {"command":"java -cp ./tools/maxent.jar density.Project ...",
  "url":"http://log.server.3:11855/jx/7090c478-ad00..."},
  ...]
```

9.5 TLQ Performance with Lifemapper

We use Lifemapper to briefly demonstrate relevant performance metrics as they apply directly TLQ. We provide details on the performance of command line queries first introduced in Chapter 6, and we investigate the overhead of executing JX queries. We compare the performance of centrally collecting all logs then querying and keeping

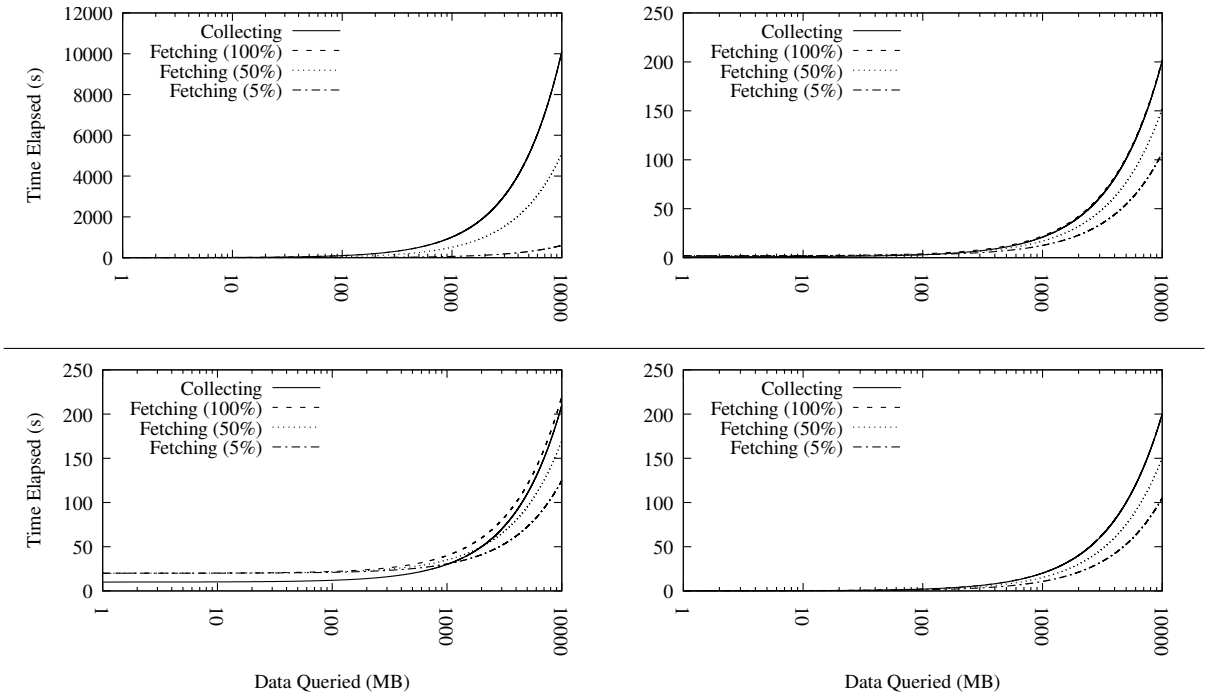

Figure 9.5. Cost of collecting compared to distributed querying. There is a scale at which it is more performant to leave logs in place rather than centralizing them. Network transfer speed (top) and query latency (bottom) affect command line queries for TLQ at different data scales.

all logs distributed, performing distributed queries.

9.5.1 Command Line Queries via TLQ

Centralizing a distributed system's debug log output has costs. Centralized systems like the Elastic Stack place all relevant data in a rendezvous node. All queries can then be executed at a single site since all data (linked or otherwise) live in the same location. As already discussed, open distributed systems make this centralizing procedure either infeasible or outright impossible. This also has overheads, primarily in moving data around. In contrast to the centralized method, data movement must be done *on demand* in an open system since we probably do not know all the locations from which we might prefetch data. This incurs an overhead at the time of

query whereas centralizing has an overhead at the time of collection. If we assume we *can* collect all the debug output of an open system to a rendezvous node, we can implement a centralized collection model of troubleshooting.

The top of Figure 9.5 shows the difference in cost between centrally collecting and keeping debug output in place performing distributed command line queries upon them (as in TLQ) with differing network speeds. We executed `grep` remotely across generated logs (each 100MB in size) hosted on 10 TLQ log servers compared to collecting all logs in a working directory and running `grep` locally [65].

9.5.2 Scalability of Command Line Queries

Often we do not receive the totality of a debug log (or logs) as the output of a command line query. We present the results of `grep` fetching 5%, 50%, and 100% of the debug logs queried to demonstrate the varying degree of overhead the distributed approach incurs compared to the collecting method. The top row of the figure shows the effect of network transfer speeds, from left to right, of 1MB/s and 100MB/s. The bottom row demonstrates the effect of query latency (the time taken to execute `grep` on each log server) on roundtrip execution time. This latency can be due to server handling and waiting, authentication overheads, etc. From left to right we have latency of 10s and 0.1s (representing 10MB/s and 1GB/s respectively). With the distributed query approach, we prevent transferring the entire log in the case when only a portion is relevant.

As the size of debug output increases, there will come a tipping point where collecting it all becomes slower than keeping logs in place. Considering only network overhead (the top row), we notice both cases of collecting only partial output of the full logs yields more performant querying than the centralized collecting method. The worst case scenario for a query would be it is so general the *whole* debug log is retrieved. This is shown in the 100% fetching line. In this worst case, we see it

matches in the slower network case and only slightly exceeds in the fast case the time taken to collect all logs and perform a local `grep`.

We notice that with high enough query latency, TLQ is actually more expensive since we may have to wait on log servers to process a user's request. The time taken for TLQ to perform distributed `grep` calls and report back was slower than directly downloading the data for local operations. However, in the more realistic low latency scenario we see a similar behavior to network overhead: when only a partial amount of each log is returned TLQ's approach is more performant.

We note there is a limit to the scalability of the collection approach since the incoming log data may saturate the network. This will leave some debug data temporarily unqueryable. Figure 6.5, previously introduced, demonstrates this effect.

9.5.3 Comparing Centralized and Distributed JX Queries

Using the JX querying functionality also displays this cutoff point where distributed queries are more costly than the centralized collection approach. Figure 9.6 shows the increasing costs of each of the four example queries from Lifemapper presented previously. Each query's cost is broken into three parts: the minimum cost if queries were evaluated at each log server and only the final output was retrieved (query), the current TLQ method where logs are fetched on demand (fetch), and the centralized approach (collect). Queries 1 and 2 fetched less than 1MB of data since the `ltrace` logs from Lifemapper are individually quite small. Query 3 fetched the Makeflow and Work Queue master debug log (which in this case is the same file for both). This single log is relatively large compared to the worker logs and `ltrace` logs. Query 4 traverses the Makeflow and master log, all worker logs, and each task trace. It fetches all the data in the system. This is equivalent in cost to the centralized approach in terms of data transferred. Table 9.1 summarizes the number of fetches and data transferred per example query.

TABLE 9.1

CENTRALIZED AND DISTRIBUTED QUERY METRICS.

Query	Fetches	Centralized Data (MB)	Distributed Data (MB)
Q1	1	278.54MB	0.03MB
Q2	2	278.54MB	0.18MB
Q3	349	278.54MB	106.22MB
Q4	1,850	278.54MB	278.54MB

In the case of Lifemapper, the benefit of keeping the logs in place exceeds the cost of centralizing only when running either redundant or successive queries. At worst, a single distributed query (such as query 4) costs only as much as centralization (purely in terms of data transferred) so long as it is not redundant (i.e. does not grab the same document multiple times). Query 5 in Figure 9.6 demonstrates this redundancy case. It is identical in output to query 4 however it redundantly asks for worker logs multiple times (an inefficiency which may arise due to poor query construction). Running multiple queries will also add up over time.

Eventually there will be a tipping point in cost, as shown when considering the combined costs of queries 1-4. Each of the examples ask different questions, as we would expect a user to do when exploring their data. Over time, with enough interesting questions, it may be that centralizing the logs would be less costly. This assumes, of course, that collecting all the logs to one machine is possible or feasible. At best there is a certain scale where the volume of logs will overwhelm the collection node. At worst, the collection node will not know about certain logs' existence nor have a means to find out about them unlike in TLQ.

All four example queries were executed with both the collection approach and

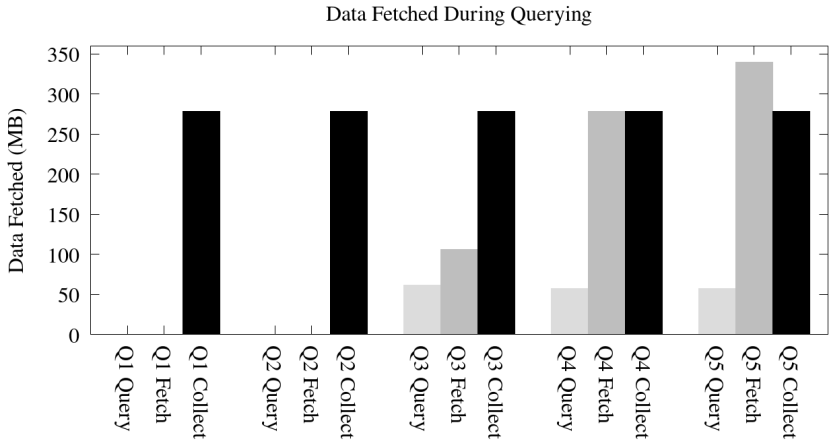

Figure 9.6. Data transferred per query. The four example queries' data transferred are presented with the calculated minimum cost if *only* the final output was returned (query), the measured distributed cost (fetch), and the measured centralized cost (collect). Query 4 is equivalent in size to centralizing the logs. Query 5 is identical to query 4 except it redundantly grabs worker logs (costing more than centralizing).

the distributed web approach. The JX evaluation process was the same for both, so each have the same number of fetches even though the centralized approach fetched locally. Centralizing logs creates an upfront cost which, in the case of Lifemapper, only pays off if the user performs many queries or runs many-hop queries as shown in example query 4. As a note, the Makeflow and Work Queue master debug log alone was 57.22MB in size.

While command line queries are extrinsic of TLQ and can thus be compared head-to-head with the collection approach, JX is more tightly coupled with the operation of TLQ. Much of the time spent executing a JX query is in the internal evaluation mechanics of JX's parser, which would be incurred in both the distributed and collection approaches equally since it would operate upon the same data and equivalent expressions. We present the performance of JX in TLQ with a few key metrics: the data usage across all log servers as compared to raw log size, the total parsing time

spent across all servers during Lifemapper's execution, and the roundtrip query time for the example queries given previously.

Among the eight log servers, 346MB were consumed for the parsers, raw logs, and JSON metalogs. 116MB of that space were the raw debug logs for the roughly $1,800$ tasks, workers, Makeflow, and the Work Queue master. The size of the parsers is negligible, 40KB. The remaining 229MB comprised the parsed JSON metalogs. We expect the JSON to be larger than the raw log since it not only contains every line of debug into a JSON array, but it also provides metadata about the component (such as its type, its links, the environment variables and files accessed, etc.).

Across the eight log servers, there was a total of $4,224$ seconds of parsing time. Recall, TLQ log servers parse logs whenever there is a change from when it last parsed the log. On log creation, the first parse is invoked. The log is then periodically re-parsed after that. On average, 528 seconds of parsing occurred at each log server while Lifemapper ran in 867 seconds. No log server spent more time parsing than it took to execute Lifemapper.

We found that simple queries (with zero or one hop to exterior links) took fractions of a second to complete in Lifemapper. The second query (Is `PATH` the same for these two tasks?) took only 0.27 seconds to complete. It is a simple lookup and also performs fetches of two quite small documents, so this makes sense. Query one (In a particular log, how many segmentation faults occurred in child processes?) took even shorter at 0.15 seconds. This is a single fetch with a regular expression evaluation upon a JSON document 48KB in size. The final query (For all workers which ran tasks for the workflow, which tasks took less than 2s?), on the other hand, is more complex than the first two. In its sentential form, we can see it is asking a question with multiple steps. In all, it took 162.79 seconds to complete. It required the traversing of every component in the system: Makeflow + the Work Queue master, each worker, and every task. All 229MB of JSON metalogs were fetched by JX, read

into memory, then evaluated upon. In systems which produce much larger logs (e.g. have particularly verbose and long-lived components), we would expect to see that queries on the whole take longer while following the trends shown in Lifemapper.

9.6 Key Usage of TLQ

We have demonstrated TLQ can be used not only in fully realized production systems; it can also be used when initially scaling up a serial workload. We took POV-Ray from an inefficient sequential pipeline to a highly parallel pipeline, and TLQ helped uncover an issue along the way. Our experience with POV-Ray highlights a few key uses for TLQ: decluttering a user's workspace on a head node (the machine they first log in to in a cluster, cloud, or grid), using command line tools directly on components in an open system, and queries uncovering interesting data for subsequent investigation.

One of the most tangible benefits of using TLQ is keeping debug logs at their machine of origin. POV-Ray debug logs are not particularly long since each frame render job is quite short. However, bringing $3,000$ logs to a user's workspace can lead to an instance of a needle in a haystack problem.

We demonstrated why using TLQ can help with the overall network overhead of the system and can reduce total roundtrip time for some queries with Lifemapper while with POV-Ray we highlighted how useful TLQ is for decluttering the workspace. In the example of the missing implicit input file, bringing all $3,000$ logs to the user would have been redundant. They all said the same thing with the same error message. We only needed to request back *one* to get that information.

Imagine, instead, that every log was different. Each one contained (mostly) unique information. Collecting all of these locally and leaving it up to the user to figure out which ones are relevant would be entirely unhelpful. It is likely they would experience information overload. Instead, with TLQ, the user chooses which information to see

and to what extent (whether running a query or retrieving the whole log).

We also demonstrated how TLQ can make effective use of already existing utilities. It provides an architecture upon which traditionally serial tools can be used effectively in an open system. The third query for POV-Ray could have used JX's `like` and `len` functions to arrive at the same answer.

But, we can also rely on proven, decades-old utilities like `grep` and `wc` which have additional built-in functionality outside the scope of a querying language. In addition, we can make use of the UNIX philosophy of pipelining multiple tools together to create a desired output. TLQ makes it possible to quickly piece together a UNIX style tool chain and execute it on arbitrary logs in an open system, something which was not possible before purely due to the user not knowing *where* their debug logs were located.

Essentially, a user is not locked in to the TLQ ecosystem. By design, TLQ does not provide its own version of popular tools. It is designed to allow the execution of these programs *on top* of its architecture. If the user wants to continue running the utilities with which they are familiar, but they now find themselves executing their workloads in an open system, TLQ empowers them to continue using what they know rather than forcing them to adopt something brand new.

Finally, we showed the importance of *exploration* in the troubleshooting process. A user typically does not know (and may not care) what information a debug log contains, its formatting, etc. *until* there is a problem. TLQ helps a user uncover the structure and data of their logs iteratively. The first POV-Ray query uncovered the set of Work Queue workers and pointed out which ones had failed tasks. The third POV-Ray query followed up on this uncovered information with a more specific question. The fourth Lifemapper query was the represents a multi-step question, the culmination of a user exploring what data exists in their logs. It is an iterative process which allows the user to explore the content of their system's debug output

at their own pace.

The queries we shown for Lifemapper were complete, and by virtue of their completeness the final two were quite complex. However, TLQ allows a user to build up to this. They do not have to know *exactly* what information they want to find from the get-go. Further, they do not have to query all the components of the system if they do not want to do so. The user can pick and choose at their own speed, iteratively building more complex and specific queries until they find the information they need. This sense of exploring the debug data rather than collecting it and having to know its structure *a priori* is a boon for users just getting started with TLQ.

CHAPTER 10

CONCLUSION

The inherent complexities of distributed systems make them notoriously difficult to effectively troubleshoot. Misbehaviors can occur in remote places in the system and ripple outward. They can also be obscured by various levels of indirection as components interact. Many tools exist to help make this process more tractable, however they lose their effectiveness when applied to *open* distributed systems. These systems have three critical features which make them more complex than general distributed systems: transient membership, on-demand placement, and resources spanning independent domains.

10.1 Summary of Key Contributions

We have introduced TLQ (Troubleshooting via Log Query) as an architecture for troubleshooting these open systems. It provides crucial functionality which allows users to effectively investigate misbehaviors in their system(s). These are log discovery, log custody, and web-inspired querying. TLQ makes the troubleshooting experience of open systems more convenient (and in some cases makes it *possible* where before it was not).

Not all problems encountered in an open system result in outright failure. We demonstrated firsthand two classes of distributed system problems (misconfigurations and performance issues) through the lens of the capacity resource provisioning metric [66]. While both of these classes *can* end in the termination of a system, we saw when experienced in Work Queue this was not the case. Implementing a basic

troubleshooting dashboard for this specific problem led to further examination of how distributed systems are structured and why troubleshooting them is so difficult.

When first implementing TLQ, we determined there were two key pieces of functionality which were necessary to overcome the additional complexities of troubleshooting an open system: log discovery and log custody. A user is not typically aware of where all the components in their system live. This is especially true of open systems where it may be impossible to transparently make the location of components known to the user without help. Log discovery in TLQ surmounts this barrier. Each component which lands on a machine being monitored by TLQ can advertise its existence to the local log server. A unique ID will be assigned to this component, acting as a name for the component's log. This name is then communicated to the user, advertising to them its existence. Now that a user knows *what* components exist and *where*, they are able to retrieve that component's debug log or operate upon it with command line tools remotely via TLQ.

But, some component are *ephemeral*. They exist for only a short time. After they complete, the underlying resource provisioning mechanisms are liable to remove all traces of its existence. Log custody enables each log server to take ownership of all logs they are told exist. Rather than logs existing in a sandbox which will be cleaned up upon a component's completion, they can be redirected to the log server's working directory. If this is not possible, it will periodically copy the log. Users are allowed to take their time analyzing or requesting individual logs since they are retained at the log servers until explicitly told otherwise. Not only does TLQ keep track and own each log. It also parses each to a standard format (in this case JSON). Not every tool can handle heterogeneous formats, so TLQ makes it possible to reasonably query any debug log using one format.

Finally, TLQ provides a querying experience designed from the beginning to operate in an open system. We discussed various existing database technologies and

a graph traversal engine, showing how each fell short of TLQ's use case. It became clear that the structure of an open distributed system is much like the World Wide Web. Components interact with each other, creating links which can be traversed to find interesting connections in the system which may have been obfuscated to the user. The JX language was expanded to give it the necessary functionality to perform queries which traverse these links. The additions to JX allow powerful queries to be constructed from a few basic operations: `fetch`, `select`, and `project`. Other built-in JX functions provide a breadth of the types of questions a user may want to ask of their debug output, and we have demonstrated examples of those interesting questions in practice as JX queries.

While the primary research outcome of our work creating TLQ was verifying its capability to accurately help troubleshoot open systems, we also explored certain performance considerations for using TLQ. These included roundtrip querying times, network data transfer overhead, cases where the performance of log servers and the user client degrade, and the choice of querying language. We demonstrated that at scale TLQ's architecture is more performant for its use case than its contemporaries which collect debug output to a centralized node (or set of nodes). Only relevant output is transferred to the user upon request, eliminating the need for expensive data transfers when possible.

All together, TLQ's mechanisms and its architecture allow us to answer significant questions about open distributed systems. When my distributed system does something unexpected, where do I look to find out why? Which pieces of the system are likely culprits for unexpected behavior? How can I access those, if I can at all? How do I make sense of their debug output? At the most fundamental level, TLQ provides transparency where the system may be obscured, gives direct access to debug output, and provides a rich querying experience designed particularly for open distributed systems.

10.2 Potential for Future Work

The creation of an interactive web interface with a visualization showing the relationship(s) between each component as a property graph would be a useful next step in TLQ's development. This would go a long way toward increasing user understanding of how their system is structured. In addition, this web interface would act as a user client through which a user could submit both JX and command line queries. The command line interface does not provide functionality like autoformatting and providing tips on how to construct JX queries. This is made practical with a web-based graphical user interface. Novel research could be conducted on the effectiveness of visualizations to help a user comprehend how their system behaves.

The second lesson learned presented in Chapter 6 focused on the lack of standardization in log structure across components. The parsers provided with TLQ went a long way to making a standardized, queryable interface for JX. However, the creation of individualized parsers for each type of debug log was a tedious task. Each parser had a significant piece of functionality in common: each transformed all lines of the original log into a JSON representation. The primary customization came from figuring out what metadata was necessary to pull out of the log to put at the top level (for ease of querying) and how to find the links each component's debug log made to other components. It may be possible to more intelligently find these links, and it may not be necessary to tediously pull out metadata if each line of debug output (as JSON) is made readily queryable. It is worth investigating whether a single, generic parser is sufficient to provide a useful, queryable set of debug output for TLQ. Further, it is worth investigating policies for parsing logs (such as on-demand when a component is queried, periodically, or with event-based mechanisms such as component completion).

Finally, a usability study of TLQ could prove interesting. Considering long-term system level tools like Recon [76] for supercomputers operate on debug logs which

accumulate data for months at a time, it would be useful to assess TLQ's capability to handle similar situations. Our work thus far has focused on using TLQ on a per-user basis. The user sets up the log servers (whether by hand or submitting them to a scheduler), modifies their system to use the monitor script on each component, and starts up the user client. However, TLQ used in a system administration capacity would accommodate potentially many simultaneous users. Both the scale of data and of users would be investigated to see the limits of TLQ.

10.3 Parting Thoughts

We live in a society exquisitely dependent on science and technology and yet have cleverly arranged things so that almost no one understands science and technology.

- Carl Sagan

Working on TLQ has provided a key insight to the nature of contemporary distributed systems. We have seen more layers of indirection being placed on components such as using a submitter process to send off function invocations to a staging node in a cloud which then forwards those computations to serverless computing resources which then operate according to their own rules. Compounding this is an increase in *things* which are becoming distributed. This includes adding edge devices and Internet of Things devices to the cloud, allowing them to be harnessed as low-power, expendable resources. Consumer workstations and laptops can be plugged in to cluster resources, supplementing the usual server rack machines used in academic and industry compute resources creating wildly heterogeneous systems. Cloud computing is becoming the norm for how services are provided. Things which were once traditionally hosted in a centralized or closed fashion have begun being served in open distributed systems.

TLQ provides a way to peel back the obfuscation these trends have placed over

components. However, it is not necessarily the *only* way this can be done. We imagine it will become a greater priority for users of distributed systems to better understand how their systems behave and why. This is becoming harder to easily provide. TLQ shows an example of how an open distributed system can be made transparent, how it can be investigated, and why it is important that users be able to interact with their systems to better understand how they work.

BIBLIOGRAPHY

1. J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza. Distributed storage and querying techniques for a semantic web of scientific workflow provenance. In *2010 IEEE International Conference on Services Computing*, pages 178–185. IEEE, 2010.

2. D. Abramson, M. N. Dinh, D. Kurniawan, B. Moench, and L. DeRose. Data centric highly parallel debugging. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 119–129. ACM, 2010.

3. V. Agrawal, D. Kotia, K. Moshirian, and M. Kim. Log-based cloud monitoring system for openstack. In *2018 IEEE Fourth International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 276–281. IEEE, 2018.

4. G. Antoniou, E. Franconi, and F. Van Harmelen. Introduction to semantic web ontology languages. In *Reasoning web*, pages 1–21. Springer, 2005.

5. K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing packet obituaries. In *ACM HotNets-III*, 2004.

6. D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–10. IEEE, 2007.

7. D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, et al. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343. ACM, 2017.

8. J. Bailey, F. Bry, T. Furche, and S. Schaffert. Web and semantic web query languages: A survey. In *Reasoning Web*, pages 35–133. Springer, 2005.

9. S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a traditional debugger to debug massively parallel applications. *Journal of Parallel and Distributed Computing*, 64(5):617–628, 2004.

10. P. Bates. Distributed debugging tools for heterogeneous distributed systems. In *[1988] Proceedings. The 8th International Conference on Distributed*, pages 308–315. IEEE, 1988.

11. P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4): 255–264, 1983.

12. A. Beguelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6):88–95, 1993.

13. D. Behrens, M. Serafini, F. P. Junqueira, S. Arnautov, and C. Fetzer. Scalable error isolation for distributed systems. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 605–620, 2015.

14. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific american*, 284(5):34–43, 2001.

15. I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Queue*, 14(2):50, 2016.

16. P. Bodic, G. Friedman, L. Biewald, H. Levine, G. Candea, K. Patel, G. Tolle, J. Hui, A. Fox, M. I. Jordan, et al. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 89–100. IEEE, 2005.

17. D. Borkar, R. Mayuram, G. Sangudi, and M. Carey. Have your data and query it too: From key-value caching to big data management. In *Proceedings of the 2016 International Conference on Management of Data*, pages 239–251. ACM, 2016.

18. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985. ISSN 0734-2071. doi: 10.1145/214451.214456. URL `http://doi.acm.org/10.1145/214451.214456`.

19. K. S.-P. Chang and S. J. Fink. Visualizing serverless cloud application logs for program understanding. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 261–265. IEEE, 2017.

20. E. Chuah, A. Jhumka, S. Alt, T. Damoulas, N. Gurumdimma, M.-C. Sawley, W. L. Barth, T. Minyard, and J. C. Browne. Enabling dependability-driven resource use and message log-analysis for cluster system diagnosis. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 317–327. IEEE, 2017.

21. F. G. de Oliveira Neto, M. Jones, and R. da Silva Martins. Visualisation to support fault localisation in distributed embedded systems within the automotive industry. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–117. IEEE, 2018.

22. E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. doi: 10.1016/j.future.2014.10.008. URL `http://pegasus.isi.edu/publications/2014/2014-fgcs-deelman.pdf`. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

23. L. DeRose, A. Gontarek, A. Vose, R. Moench, D. Abramson, M. N. Dinh, and C. Jin. Relative debugging for a highly parallel hybrid computer system. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

24. J. DeSouza, B. Kuhn, B. R. De Supinski, V. Samofalov, S. Zheltov, and S. Bratanov. Automated, scalable debugging of mpi programs with intel® message checker. In *Proceedings of the second international workshop on software engineering for high performance computing system applications*, pages 78–82. ACM, 2005.

25. N. Dryden. Pgdb: A debugger for mpi applications. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, page 44. ACM, 2014.

26. Ú. Erlingsson, M. Peinado, S. Peter, M. Budiu, and G. Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):13, 2012.

27. M. Feng, L. Tan, and R. Gupta. Lightweight fault detection in parallelized programs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE Computer Society, 2013.

28. E. Fromentin, N. Plouzeau, and M. Raynal. An introduction to the analysis and debug of distributed computations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 2, pages 545–553. IEEE, 1995.

29. T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. Rdf querying: Language constructs and evaluation methods compared. In *Reasoning Web International Summer School*, pages 1–52. Springer, 2006.

30. H. Garcia-Molina, F. Germano, and W. H. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, (2):210–219, 1984.

31. A. Gehani, M. Kim, and T. Malik. Efficient querying of distributed provenance stores. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 613–621. ACM, 2010.

32. O. Gerstel, S. Zaks, M. Hurfin, N. Plouzeau, and M. Raynal. On-the-fly replay: a practical paradigm and its implementation for distributed debugging. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*, pages 266–272. IEEE, 1994.

33. T. Godin, M. J. Quinn, and C. M. Pancake. Parallel performance visualization using moments of utilization data. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 777–782. IEEE, 1998.

34. M. A. Gulzar. Interactive and automated debugging for big data analytics. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 509–511. ACM, 2018.

35. M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 520–534. ACM, 2017.

36. H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 7:1–7:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670986. URL `http://doi.acm.org/10.1145/2670979.2670986`.

37. J. Gustafson, G. Montry, and R. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9(4), 1988.

38. J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL `http://doi.acm.org/10.1145/42411.42415`.

39. C. Gyorodi, R. Gyorodi, G. Pecherle, and A. Olah. A comparative study: Mongodb vs. mysql. In *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, pages 1–6, June 2015. doi: 10.1109/EMES.2015.7158433.

40. T. Hacker, R. Pais, and C. Rong. A markov random field based approach for analyzing supercomputer system logs. *IEEE Transactions on Cloud Computing*, 2017.

41. N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 55–60. ACM, 2012.

42. N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.

43. O. Hartig and J. Pérez. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1155–1164, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee. ISBN 978-1-4503-5639-8. doi: 10.1145/3178876.3186014. URL `https://doi.org/10.1145/3178876.3186014`.

44. C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.

45. S. He, J. Zhu, P. He, and M. R. Lyu. Experience report: system log analysis for anomaly detection. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*, pages 207–218. IEEE, 2016.

46. M. T. Heath and J. A. . Etheridge. Visualizing the performance of parallel programs. *IEEE software*, 8(5):29–39, 1991.

47. I. Horrocks, P. F. Patel-Schneider, and F. Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of web semantics*, 1(1): 7–26, 2003.

48. P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 1–16, 2018.

49. K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer. State of the art of performance visualization. *EuroVis 2014*, 2014.

50. N. Jamadagni and Y. Simmhan. Godb: From batch processing to distributed querying over property graphs. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 281–290. IEEE, 2016.

51. M. Jarke and Y. Vassiliou. A framework for choosing a database query language. In *Readings in Artificial Intelligence and Databases*, pages 363–375. Elsevier, 1989.

52. N. B. Jensen, N. Q. Nielsen, G. L. Lee, S. Karlsson, M. Legendre, M. Schulz, and D. H. Ahn. A scalable prescriptive parallel debugging model. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 473–483. IEEE, 2015.

53. Z. Jia, C. Shen, X. Yi, Y. Chen, T. Yu, and X. Guan. Big-data analysis of multi-source logs for anomaly detection on network-based system. In *2017 13th IEEE Conference on Automation Science and Engineering (CASE)*, pages 1136–1141. IEEE, 2017.

54. E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

55. S. T. Jones, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, et al. Antfarm: Tracking processes in a virtual machine environment. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2006.

56. J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 5(2):121–150, 1987.

57. A. S. Kanade and A. Gopal. Choosing right database system: Row or column-store. In *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pages 16–20. IEEE, 2013.

58. N. Khadke, M. P. Kasick, S. P. Kavulya, J. Tan, and P. Narasimhan. Transparent system call based performance debugging for cloud computing. In *Presented as part of the 2012 Workshop on Managing Systems Automatically and Dynamically*, 2012.

59. A. Khalid, J. J. Quinlan, and C. J. Sreenan. Mininam: A network animator for visualizing real-time packet flows in mininet. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 229–231. IEEE, 2017.

60. M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112. ACM, 2008.

61. E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 191–203. ACM, 2008.

62. D. Kranzlmüller. Scalable parallel program debugging with process isolation and grouping. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 294. IEEE Computer Society, 2002.

63. D. Kranzlmüller, S. Grabner, and J. Volkert. Event graph visualization for debugging large applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 108–117. ACM, 1996.

64. D. Kranzlmüller, S. Grabner, and J. Volkert. Debugging with the mad environment. *Parallel Computing*, 23(1-2):199–217, 1997.

65. N. Kremer-Herman and D. Thain. Log discovery for troubleshooting open distributed systems with tlq. In *Practice and Experience in Advanced Research Computing*, PEARC '20, page 224–231, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450366892. doi: 10.1145/3311790.3396633. URL `https://doi.org/10.1145/3311790.3396633`.

66. N. Kremer-Herman, B. Tovar, and D. Thain. A lightweight model for right-sizing master-worker applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 39:1–39:13, Piscataway, NJ, USA, 2018. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=3291656.3291708`.

67. S. Krishnaprasad. Uses and abuses of amdahl's law. *J. Comput. Sci. Coll.*, 17 (2):288–293, Dec. 2001. ISSN 1937-4771. URL `http://dl.acm.org/citation.cfm?id=775339.775386`.

68. A. Lahmadi and F. Beck. Powering Monitoring Analytics with ELK stack, June 2015. URL `https://hal.inria.fr/hal-01212015`.

69. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL `http://doi.acm.org/10.1145/359545.359563`.

70. L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

71. L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

72. L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

73. M. Lanthaler and C. Gütl. On using json-ld to create evolvable restful services. In *Proceedings of the Third International Workshop on RESTful Design*, pages 25–32, 2012.

74. T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. Technical report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1986.

75. G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. De Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–9. IEEE, 2008.

76. K. H. Lee, N. Sumner, X. Zhang, and P. Eugster. Unified debugging of distributed systems with recon. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 85–96. IEEE, 2011.

77. T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. {SAMC}: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 399–414, 2014.

78. J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 279–294. ACM, 2011.

79. Q. Liao, A. Blaich, A. Striegel, and D. Thain. Enavis: Enterprise network activities visualization. In *LISA*, pages 59–74, 2008.

80. W. Lin, C. Krintz, and R. Wolski. Tracing function dependencies across clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260, July 2018. doi: 10.1109/CLOUD.2018.00039.

81. X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. 2008.

82. D. Logothetis, S. De, and K. Yocum. Scalable lineage capture for debugging disc analytics. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 17. ACM, 2013.

83. J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 35 (4):11, 2018.

84. E. Maguire, P. Rocca-Serra, S.-A. Sansone, J. Davies, and M. Chen. Visual compression of workflow visualizations with automated detection of macro motifs. *IEEE transactions on visualization and computer graphics*, 19(12):2576–2585, 2013.

85. S. Marr, C. Torres Lopez, D. Aumayr, E. Gonzalez Boix, and H. Mössenböck. A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools. In *ACM SIGPLAN Notices*, volume 52, pages 3–14. ACM, 2017.

86. H. Matsuba, M. Hiltunen, K. Joshi, and R. Schlichting. Discovering the structure of cloud applications using sampled packet traces. In *2014 IEEE International Conference on Cloud Engineering*, pages 235–244. IEEE, 2014.

87. E. Miller. An introduction to the resource description framework. *Bulletin of the American Society for Information Science and Technology*, 25(1):15–19, 1998. doi: 10.1002/bult.105. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/bult.105`.

88. T. Morsellino, C. Aguerre, and M. Mosbah. Debugging the execution of distributed algorithms over anonymous networks. In *2012 16th International Conference on Information Visualisation*, pages 464–470. IEEE, 2012.

89. H. Nguyen, Y. Tan, and X. Gu. Pal: Propagation-aware anomaly localization for cloud hosted distributed applications. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, page 1. ACM, 2011.

90. L. Oldenburg, X. Zhu, K. Ramasubramanian, and P. Alvaro. Fixed it for you: Protocol repair using lineage graphs. In *CIDR*, 2019.

91. A. J. Oliner and A. Aiken. A query language for understanding component interactions in production systems. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 201–210. ACM, 2010.

92. M. Owens. *The definitive guide to SQLite*. Apress, 2006.

93. X. Pan, J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Ganesha: Blackbox diagnosis of mapreduce systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(3):8–13, 2010.

94. C. M. Pancake. Debugger visualization techniques for parallel architectures. In *Digest of Papers COMPCON Spring 1992*, pages 276–284. IEEE, 1992.

95. C. M. Pancake and C. Cook. What users need in parallel tool support: Survey results and analysis. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 40–47. IEEE, 1994.

96. C. M. Pancake and R. H. Netzer. A bibliography of parallel debuggers. In *ACM SIGPLAN Notices*, volume 28, pages 169–186. ACM, 1993.

97. C. M. Pancake and S. Utter. Models for visualization in parallel debuggers. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 627–636. ACM, 1989.

98. Z. Parker, S. Poe, and S. V. Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1901-0. doi: 10.1145/2498328.2500047. URL `http://doi.acm.org/10.1145/2498328.2500047`.

99. M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

100. C. Pham, L. Wang, B. C. Tak, S. Baset, C. Tang, Z. Kalbarczyk, and R. K. Iyer. Failure diagnosis for distributed systems using targeted fault injection. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):503–516, 2016.

101. W. Puangsaijai and S. Puntheeranurak. A comparative study of relational database and key-value database for big data applications. In *2017 International Electrical Engineering Congress (iEECON)*, pages 1–4. IEEE, 2017.

102. T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 472–484. ACM, 2010.

103. A. Rabkin and R. Katz. Precomputing possible configuration error diagnoses. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 193–202. IEEE Computer Society, 2011.

104. A. Rabkin and R. H. Katz. How hadoop clusters break. *IEEE software*, 30(4): 88–94, 2013.

105. D. Rajan, A. Thrasher, B. Abdul-Wahid, J. A. Izaguirre, S. Emrich, and D. Thain. Case studies in designing elastic applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 466–473, May 2013. doi: 10.1109/CCGrid.2013.46.

106. P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, volume 6, pages 9–9, 2006.

107. R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger. So, you want to trace your distributed system? key design insights from years of practical experience. *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14*, 2014.

108. R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger. Principled workflow-centric tracing of distributed systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 401–414. ACM, 2016.

109. K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau. Noise injection techniques to expose subtle and unintended message races. In *ACM SIGPLAN Notices*, volume 52, pages 89–101. ACM, 2017.

110. J. Scholten and P. Jansen. Distributed debugging and tumult. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, pages 172–176. IEEE, 1990.

111. S. Schulz and C. Bockisch. A blast from the past: online time-travel debugging with bite. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, page 13. ACM, 2018.

112. C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 291–309, 2016.

113. N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

114. T. Shaffer, N. Kremer-Herman, and D. Thain. Flexible partitioning of scientific workflows using the jx workflow language. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372275. doi: 10.1145/3332186.3338100. URL `https://doi.org/10.1145/3332186.3338100`.

115. K. Shibanai and T. Watanabe. Actoverse: a reversible debugger for actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 50–57. ACM, 2017.

116. K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

117. B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

118. A. Singh, P. Maniatis, T. Roscoe, and P. Druschel. Using queries for distributed monitoring and forensics. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 389–402. ACM, 2006.

119. S. Sistare, D. Allen, R. Bowker, K. Jourdenais, J. Simons, and R. Title. A scalable debugger for massively parallel message-passing programs. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 825–832. IEEE, 1994.

120. A. Spear, M. Levy, and M. Desnoyers. Using tracing to solve the multicore system debug problem. *Computer*, 45(12):60–64, 2012.

121. M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström. Json-ld 1.0. *W3C Recommendation*, 16:41, 2014.

122. W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1887–1901. ACM, 2015.

123. B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan. Logan: Problem diagnosis in the cloud using log-based reference models. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*, pages 62–67. IEEE, 2016.

124. J. Tan, X. Pan, S. Kavulya, R. Ghandi, and P. Narasimhan. Mochi: visual log-analysis based tools for debugging hadoop. 2009.

125. J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *2010 IEEE 30th International Conference on Distributed Computing Systems*, pages 795–806. IEEE, 2010.

126. A. Tarafdar and V. K. Garg. Debugging in a distributed world: observation and control. In *Proceedings. 1998 IEEE Workshop on Application-Specific Software Engineering and Technology. ASSET-98 (Cat. No. 98EX183)*, pages 151–156. IEEE, 1998.

127. D. Thain and C. Moretti. Abstractions for cloud computing with condor. *Cloud Computing and Software Services: Theory and Techniques*, pages 153–171, 2010.

128. X. Tu, H. Jin, X. Fan, and J. Ye. Meld: A real-time message logic debugging system for distributed systems. In *2010 IEEE Asia-Pacific Services Computing Conference*, pages 59–66. IEEE, 2010.

129. J. Turnbull. *The Logstash Book*. James Turnbull, 2013.

130. O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, pages 7:1–7:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4780-8. doi: 10.1145/2960414.2960421. URL `http://doi.acm.org/10.1145/2960414.2960421`.

131. S. Venkatesan and B. Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2): 163–177, 1995.

132. L. Walsh, V. Akhmechet, and M. Glukhovsky. Rethinkdb-rethinking database storage, 2009.

133. P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen. Cloudranger: root cause identification for cloud native systems. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 492–502. IEEE Press, 2018.

134. S. Weigert, M. Hiltunen, and C. Fetzer. Mining large distributed log data in near real time. In *Managing Large-scale Systems via the Analysis of System Logs and the Application of Machine Learning Techniques*, page 5. ACM, 2011.

135. S. Whitlock, C. Scott, and S. Shenker. Brief announcement: techniques for programmatically troubleshooting distributed systems. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 134–136. ACM, 2013.

136. M. Whittaker, C. Teodoropol, P. Alvaro, and J. M. Hellerstein. Debugging distributed systems with why-across-time provenance. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 333–346. ACM, 2018.

137. M. C. Wikstrom and J. L. Gustafson. The twin bottleneck effect. In *[1993] Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences*, volume ii, pages 574–583 vol.2, Jan 1993. doi: 10.1109/HICSS.1993.284068.

138. Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems with negative provenance. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 383–394. ACM, 2014.

139. T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

140. T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015.

141. T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *OSDI*, pages 619–634, 2016.

142. K. Yamnual, P. Phunchongharn, and T. Achalakul. Failure detection through monitoring of the scientific distributed system. In *2017 International Conference on Applied System Innovation (ICASI)*, pages 568–571, May 2017. doi: 10.1109/ICASI.2017.7988485.

143. L. Yu. *Right-sizing Resource Allocations for Scientific Applications in Clusters, Grids, and Clouds*. PhD thesis, University of Notre Dame, 2013.

144. D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010.

145. D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 249–265, 2014.

146. J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *ACM SIGPLAN Notices*, volume 49, pages 687–700. ACM, 2014.

147. Q. Zhang, D. Yan, and J. Cheng. Quegel: A general-purpose system for querying big graphs. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2189–2192. ACM, 2016.

148. W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 615–626. ACM, 2010.

149. W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 295–310. ACM, 2011.

150. Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. Netcheck: Network diagnoses from blackbox traces. In *NSDI*, pages 115–128, 2014.