Log Discovery for Troubleshooting Open Distributed Systems with TLQ

NATHANIEL KREMER-HERMAN, University of Notre Dame
DOUGLAS THAIN, University of Notre Dame

Troubleshooting a distributed system can be incredibly difficult. 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, 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. However, 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 failures these logs have recorded because the location of those logs is usually not transparent to the user (and by association the troubleshooting tools they are using). We present TLQ, a framework designed from first principles for log discovery to enable troubleshooting of 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 concretely 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.

ACM Reference Format:
Nathaniel Kremer-Herman and Douglas Thain. 2020. Log Discovery for Troubleshooting Open Distributed Systems with TLQ. In *Practice and Experience in Advanced Research Computing 2020, July 26–30, 2020, Portland, OR.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 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 failures 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 a failure on their workstation, it can be much more difficult and intimidating to troubleshoot failures 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 computations

Manuscript submitted to ACM

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).

From first principles, we have encountered the 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) [14, 20]. 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).

From this observation, we have designed from first principles TLQ (Troubleshooting via Log Query), a framework for log discovery and troubleshooting 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 location 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 using the Lifemapper application. 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 TLQ.

2 CHALLENGES OF DISTRIBUTED TROUBLESHOOTING

To demonstrate why distributed troubleshooting is so complex, consider a typical scientific workflow application. The workflow management system is executing its workflow via a master-worker execution framework. Figure 1 demonstrates the architecture of this distributed system. The workflow manager sends tasks (i.e. definitions of work to be done along with inputs and expected outputs) to the master process of the master-worker execution framework. This master process then dispatches the task's command and inputs to a chosen worker process on some other compute node. It does this for each task the workflow manager gives it (assume some considerable scale such as 10,000 tasks). Each

[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.

worker runs the received command which uses the inputs to (hopefully) generate the expected outputs. The specified inputs and outputs for each task are stored in a shared cache at the worker for future use. Because the application is a scientific workflow, there exist data dependencies between tasks, so tasks are submitted by the workflow manager only when their respective input dependencies are satisfied.

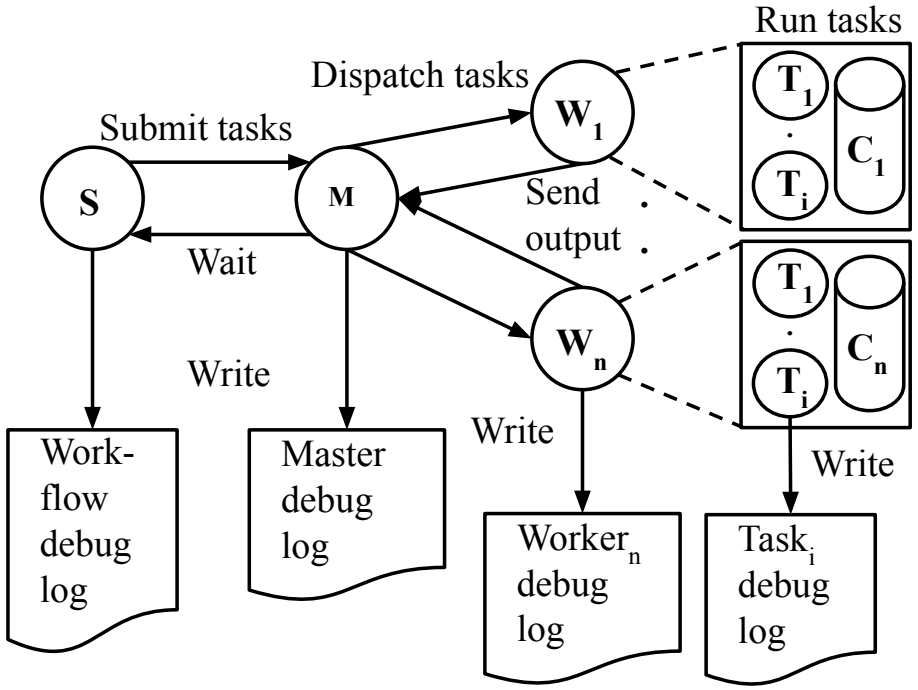

Fig. 1. Scientific Workflow Architecture. The workflow manager (S) submits tasks to the master process (M). The master dispatches tasks to workers (W). Workers run tasks (T) locally. Each component writes to its own log file.

Failures can occur in a scientific workflow due to the way the workflow manager is designed. In this notional example, it is assumed the workflow manager dispatches new work when data dependencies are satisfied. This is determined via file existence, a common method for doing so. Because of this design, it is not uncommon for a task to *silently* fail while still producing *some* of its expected output. Suppose this failure occurs after the expected output has been created by a task (task `t`), but that output is malformed. The tasks which depend upon the successful creation of this malformed output would in turn fail. Because a scientific workflow is only successful when the final output is created, the entire workflow would fail in turn.

How might the user uncover this failure? They may have some monitoring dashboard, use command line utilities like `grep`, or seek through files manually. Regardless their tools, they most likely will only be given a piece of the puzzle for finding the cause of failure. They will need to uncover more and more pieces as they dissect their system. This becomes an intractable method at larger scales.

Beginning with the perspective of the workflow management system's log, it would appear that task `t` *did* create its expected output. So, the workflow management system would consider `t` a success. The troubleshooting tools used (or manual inspection performed) upon the log will not detect any explicit error for task `t` here. The same is true of the

worker process on which the task executed. It sees the specified output is created. According to both these logs, the task was a success. This is because the workflow manager and worker check for file *existence* rather than file *correctness*. It is often infeasible to require they do more than that unless the manager is written for a specific domain or narrow set of applications. However, the workflow manager and the worker logs give only a fragment of the larger story for this failure. These two logs, while helpful in indicating possible components to investigate, do not yield the cause of failure.

The user knows that during execution of this notional example, all of task `t`'s descendants fail according to the workflow manager and worker logs. So, they would likely next investigate the explicitly failed tasks' logs to figure out how they may have failed. From their perspective, it is still entirely possible that the code executing on these tasks is incorrect rather than task `t` (which *silently* failed). Upon inspection through either tools or manual scanning, they could see that each of these tasks failed shortly after opening the malformed output file of task `t`.

It is at this point the user has uncovered enough evidence to go to task `t`'s log. Depending on their technical knowledge, they may be able to tell from the log that task `t` failed silently during its execution. They may see a similar pattern to the dependent tasks where the output file is opened for reading, then the program abruptly fails. The user can then look at the contents of that output file to find that the data was not completely written before the task ended. At this point, the problem is now a serial debugging issue rather than a distributed troubleshooting problem now that the specific component at fault has been identified. The user can now run this task locally to debug the specific low-level reasons why the program it runs fails early (and silently).

This experience is clunky to the user since the onus is on them to know *which* logs to read and *where* each log resides. Monitoring dashboards or other commonplace tools reduce the overall time spent investigating all these logs, however they can only enable troubleshooting if the locations of each log are known. In this notional example, we assume the user can find each log. This is not always the case, however, since the user does not typically decide where their components execute (e.g. using a batch system to schedule worker processes) nor may they necessarily be explicitly told after the fact where those components were sent.

3 TROUBLESHOOTING AS DISTRIBUTED QUERYING

The notional example highlights a few key insights as to why troubleshooting distributed systems is so difficult:

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

In the notional example 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 be 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).

We demonstrated that the silent failure of task `t` did not present itself until its descendants failed explicitly. Making connections between `t` and its descendant tasks is difficult if the user does not know the relationship between these tasks beforehand *and* if that relationship is not made clear in the logs. Moreover, finding the cause in the notional

example was made possible because we assumed the user knew the name of the malformed output from `t` which allowed them to understand the relationship between it and the failed tasks (often not the case with very large systems).

In the notional example, 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 failure. Further, each log only provided pieces of the cause of failure until the user found task `t`'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 matters which sends to computation 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 failures *without* having to know which computations happened where).

3.1 Querying Logs in Place Across Domains

Rather than transferring debug output to a centralized node as is done in current state of the art [14, 20], we keep each log in place. This is due to both performance reasons and to enable us to troubleshoot an *open* distributed system. The debug output may be too large to reasonably transfer to a centralized rendezvous node (or may cost too much to transfer from cloud storage), but that output may *also* exist across autonomous domains such that transferring them all to a single node is impossible due to policy restrictions from each domain's administrators. Further, it is likely that while the system knows the logs being produced, that information is not transparent to the user. Often when using a centralized rendezvous node, the user annotates which files to transfer. If they do not know these files' names, that is difficult to make happen.

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 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 upon the servers' tracked logs or a request

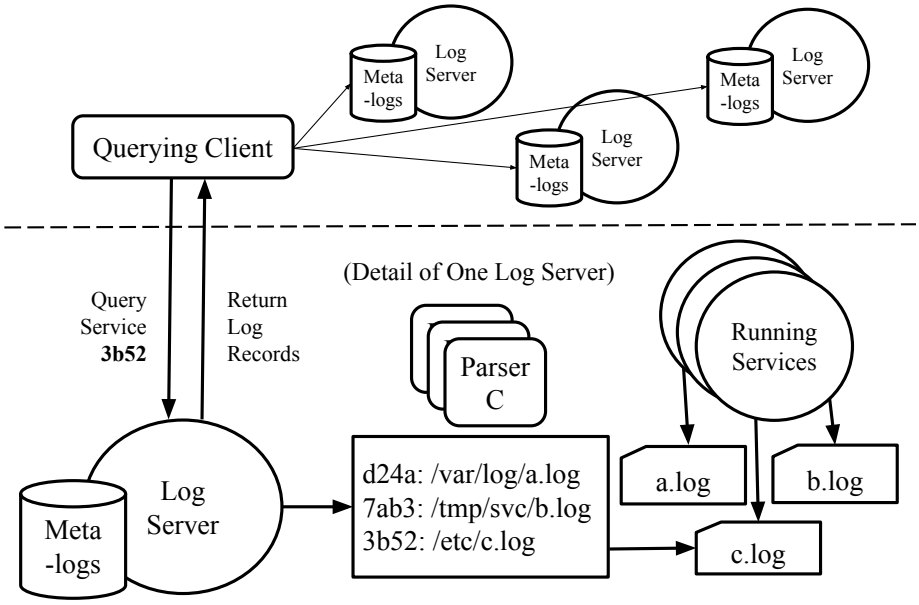

Fig. 2. TLQ System Architecture. TLQ queries are either invocations of a troubleshooting tool or a request for a particular log. Each log server manages a set of parsers that consume local log files for key information and export it to JSON documents to facilitate troubleshooting.

to transfer a specific log for manual inspection (if possible). The set of log watchers are discoverable end points for an open system since their existence is advertised to the user. Figure 2 demonstrates the architecture of TLQ in action.

4 IMPLEMENTATION

The underlying troubleshooting system architecture is composed of four parts: a Perl log watch HTTP server, a set of log parsers, a Perl querying client, and a wrapper script which communicates with the log watcher and user client. Refer to Figure 2 for an expanded view of this architecture. The Perl 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. 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 server running alongside the client. A wrapper shell 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 in question.

The wrapper 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 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 wrapper script waits for the server to update this list with the IDs of the files the wrapper 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 wrapper then executes the command it had wrapped, starting up a component of the system.

5 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). TLQ provides the log discovery mechanism and the capability to run `grep` at each log watcher by way of TLQ's 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.

Lifemapper is a biodiversity scientific workflow executed using the Makeflow workflow management system [17]. 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 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 to finish.

We made use of the Work Queue master-worker execution engine to run the Makeflow rules. Figure 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 (a roughly equivalent definition 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 failures arise. The workflow management system, master process, and batch

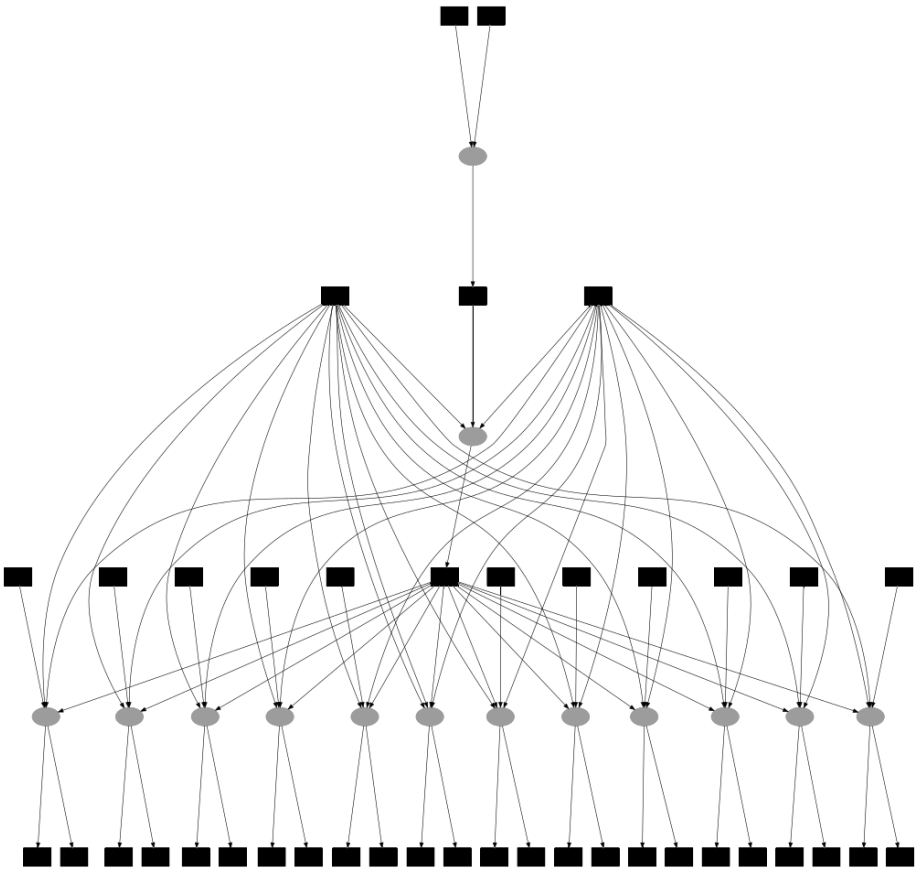

Fig. 3. Structure of Lifemapper showing data dependencies. Squares represent files, and circles represent processes. Its structure allows for a high degree of parallelism.

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 [13]).

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 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 wrapper script wrapped around each component in the system). We find it, and through the TLQ client we query the relevant

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

Table 1. Query Roundtrip Time for Lifemapper.

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) because multiple threads of the command encountered segmentation faults due to the unhandled exception:

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

Table 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 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.

5.1 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. 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 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

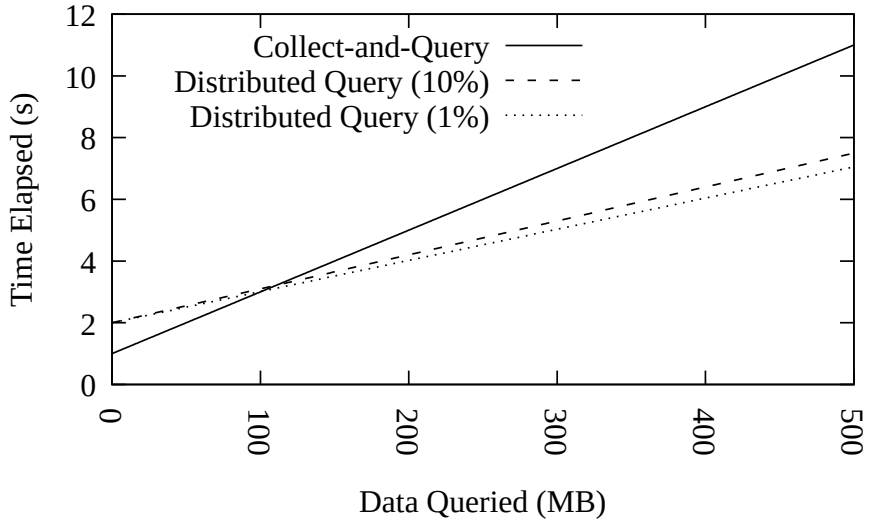

Fig. 4. Cost of collecting and querying compared to distributed querying. All things being equal (communication overhead, transfer speed, and local read speed) there is a scale at which distributed queries are faster than the collect-and-query approach used by centralized architectures.

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 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 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 the log watcher node's disk was under heavy load.

Figure 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). In previous work, we have found that there is some maximum effective scale of a system called its capacity [13], and adding log data transfers further erodes the total capacity of a system. We see this in play when the system reaches

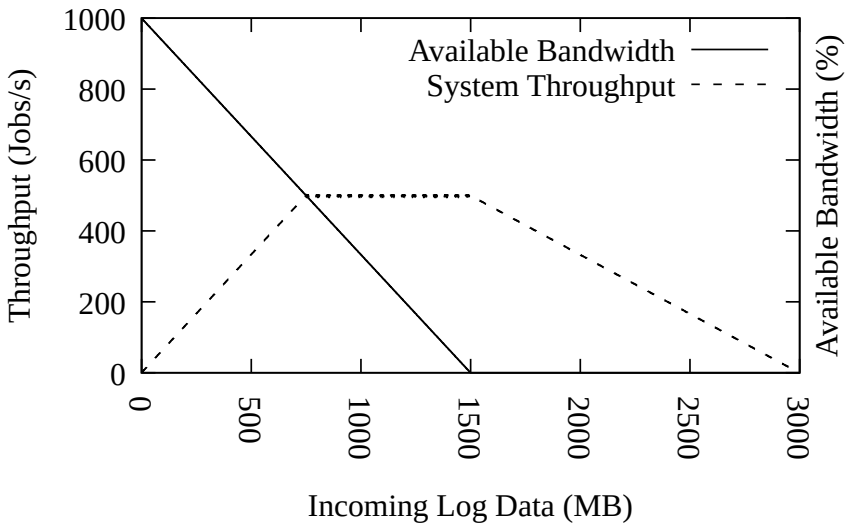

Fig. 5. Effect of centralizing log collection on system throughput. There is some scale at which centrally collecting all logs degrades system throughput due to unavailable bandwidth (i.e. the system spends so much time transmitting log and output data that it cannot do anything else).

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 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 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 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. The current state of troubleshooting and querying tools which can be added atop TLQ's software stack does not yet provide this, and we see this as a potentially information-rich aspect of distributed systems troubleshooting to be explored further.

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. In this work, 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 tool chaining as discussed previously.

In addition, there would be no need for specialized parsing if logs shared a common, transactional format. This format, possibly in 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. Having a large body of independent developers adopt a common debug log format like this, however, is wishful thinking currently.

7 RELATED WORK

There are perennial problems in distributed computing which make troubleshooting distributed systems more difficult than troubleshooting their serial counterparts. There is no common time scale between compute nodes, so many problems arise when trying to create a single history of events across a distributed system [15]. On a related note, it is also difficult to produce a snapshot of the state of a distributed system [5]. There are many other complications to troubleshooting distributed systems namely heterogeneity, concurrency, distributed state, and partial failures covered in [3]. Some of these problems have been addressed in earlier works. While many focused on low-level debugging [16, 18] of specific bugs in applications, TLQ focuses on higher-level environment interactions of a system which is more akin to [2] while leaving the more fine-grained debugging to the user's choice of tools.

There exist many tools to troubleshoot distributed applications. We note only a few here to capture the breadth of the field. The Pegasus workflow management system [7] has its own native troubleshooting tools. It has a log analyzer and web tool which can provide aggregated metrics. It can also pull up specific debug logs for the user on demand. TLQ builds upon this kind of approach, allowing a user to query the total collection of known logs such that they do not have to investigate debug logs one-by-one unless as a last resort. There are also network troubleshooting tools such as [9] which proposes the tracing of packet histories much like we provide the set of environment interactions for each component. There exist plenty of recent tools which perform low-level debugging of distributed systems and applications [8, 11, 12]. Another tool provides troubleshooting of resource provisioning issues [13] using a dashboard. Finally, another tool using code injection allows for the creation of a minimum set of events needed to recreate a known state in a distributed system [19]. Each of these approaches give users different methods of diagnosing failures in their systems, however they are not designed for *open* distributed systems. Invoking these tools as distributed queries with TLQ allows their benefits to be reaped in open systems.

In addition to various monitoring dashboards and interactive debugging tools, there exists a large body of log analyzers as well. Anomaly detection [10] is important for high-volume logging. Other works provide diagnosis capability focusing on resource usage logs [6] and network logs [21]. TLQ adds in an emphasis on environment interactions which can bolster the analysis of these two approaches. A particularly relevant tool is ShiViz [1, 4] which provides vector timestamps and space-time diagrams about events which occur in a distributed application. TLQ's JSON metalogs provide context for how components may be related or have related environments. Invoking log analyzers like ShiViz atop TLQ can make these relationships more concrete in terms of *when* relationships are formed. Most similar to our approach in terms of architecture and capability is the Elastic Stack (formerly ELK stack) consisting of the Elasticsearch, Logstash, and Kibana tools. Their use has been shown in [14, 20]. TLQ, while having a similarly modular capability of adding more tools to its software stack, differs in that a distributed query does not require continuous streaming of logs to a centralized rendezvous point as in Logstash. The query moves to the data rather than the other way around.

8 CONCLUSIONS

We introduced TLQ, an architecture which enables log discovery for *open* distributed systems. Because of the complexities of open distributed systems, it is not feasible to establish the name and location of all the debug output of that system *a priori*. This information is determined at runtime. Instead, each debug log must be discovered and its existence must be advertised to users of the system. This is made possible with TLQ. By making log locations transparent to a user, they can analyze their logs with the troubleshooting tools of their choice which are executed as queries in TLQ's client. We

demonstrated the necessity of an architecture like TLQ through a notional example, a real-world scientific workflow in action, and two models which demonstrate observations about open distributed systems which makes centralized approaches to log collection and discovery infeasible at certain scales. We also note three key lessons learned about troubleshooting open distributed systems which became apparent during the design and implementation of TLQ. In the future, we plan to provide further functionality to TLQ. Namely, we seek to implement an effective JSON querying language which will enable a user to quickly ask questions of the JSON metalogs, add interactive visualizations to a web-based TLQ client, and to add the concept of *log custody* to TLQ (where the TLQ log watchers take active ownership of the debug logs produced, thus keeping them alive after ephemeral components finish execution).

9 ACKNOWLEDGMENTS AND AVAILABILITY

This work was supported by National Science Foundation grant ACI-1642409. The source code is distributed under the GNU General Public License. The following URLs provide the source code and examples used in this work:

`github.com/cooperative-computing-lab/tlq/tree/pearc`

`github.com/cooperative-computing-lab/makeflow-examples`

REFERENCES

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D. Ernst. 2014. Shedding Light on Distributed System Executions. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. ACM, New York, NY, USA, 598–599. https://doi.org/10.1145/2591062.2591134

[2] Peter C Bates and Jack C Wileden. 1983. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software* 3, 4 (1983), 255–264.

[3] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D Ernst. 2016. Debugging distributed systems. *Queue* 14, 2 (2016), 50.

[4] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging Distributed Systems. *Commun. ACM* 59, 8 (July 2016), 32–37. https://doi.org/10.1145/2909480

[5] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[6] Edward Chuah, Arshad Jhumka, Samantha Alt, Theo Damoulas, Nentawe Gurumdimma, Marie-Christine Sawley, William L Barth, Tommy Minyard, and James C Browne. 2017. Enabling Dependability-Driven Resource Use and Message Log-Analysis for Cluster System Diagnosis. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 317–327.

[7] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. 2015. Pegasus: a Workflow Management System for Science Automation. *Future Generation Computer Systems* 46 (2015), 17–35. https://doi.org/10.1016/j.future.2014.10.008 Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[8] Nikoli Dryden. 2014. Pgdb: A debugger for mpi applications. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 44.

[9] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks.. In *NSDI*, Vol. 14. 71–85.

[10] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. Experience report: system log analysis for anomaly detection. In *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE, 207–218.

[11] Stephen T Jones, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, et al. 2006. Antfarm: Tracking Processes in a Virtual Machine Environment.. In *USENIX Annual Technical Conference, General Track*. 1–14.

[12] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F Abdelzaher, and Jiawei Han. 2008. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 99–112.

[13] Nathaniel Kremer-Herman, Benjamin Tovar, and Douglas Thain. 2018. A Lightweight Model for Right-sizing Master-worker Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 39, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291708

[14] Abdelkader Lahmadi and Frédéric Beck. 2015. Powering Monitoring Analytics with ELK stack. https://hal.inria.fr/hal-01212015

[15] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[16] Johan Scholten and PG Jansen. 1990. Distributed debugging and Tumult. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*. IEEE, 172–176.

[17] Tim Shaffer, Nathaniel Kremer-Herman, and Douglas Thain. 2019. 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)* (Chicago, IL, USA) *(PEARC '19)*. Association for Computing Machinery, New York, NY, USA, Article 103, 8 pages. https://doi.org/10.1145/3332186.3338100

[18] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. 1994. A scalable debugger for massively parallel message-passing programs. In *Proceedings of IEEE Scalable High Performance Computing Conference*. IEEE, 825–832.

[19] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 333–346.

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

[21] Yanyan Zhuang, Eleni Gessiou, Steven Portzer, Fraida Fund, Monzur Muhammad, Ivan Beschastnikh, and Justin Cappos. 2014. NetCheck: Network Diagnoses from Blackbox Traces.. In *NSDI*. 115–128.