Profiling Distributed Systems: Two Case Studies

Nelson Liu
Stanford University

Abstract

We present two case studies in profiling distributed systems to better understand their bottlenecks and hot spots.

Our first case study focuses on CodaLab, a platform for reproducible research. CodaLab is deployed as a collection of Dockerized microservices that must communicate with each other to fulfill user requests. We trace real-world user queries through the application to better understand where latency in the application arises. We find that a significant amount of latency comes from database operations, and provide some future recommendations for improving performance.

The second case study focuses on a replicated SQLite database (`rqlite`), where consensus across nodes (via Raft; [1]) is required to serve and execute user queries in a fault-tolerant manner. We measure the overhead from replication, the average and tail speed of the `AppendEntries` and `RequestVote` RPCs, finding that replication slightly affects throughput even in our small-scale benchmark.

1 Introduction

Understanding the performance and bottlenecks of traditional applications is relatively well-studied, with many mature profiling tools and workflows available for a wide variety of programming languages. However, profiling and understanding the performance of distributed applications is not as straightforward—a single user request may touch multiple services before returning a result to the user. In this increasingly-common setting, it is diffcult to disentangle the contribution of individual services on overall request latency.

To fulfill these needs, a variety of tools have been built for tracing requests through distributed systems. In this work, we present case studies of tracing for two very different distributed systems. The first case study focuses on CodaLab, [1] an end-to-end platform for reproducible research. The second case study traces requests in `rqlite`,[2] a distributed SQLite database. Through tracing, we can better understand the latency breakdown of requests through these systems and also derive useful insights for improving their throughput.

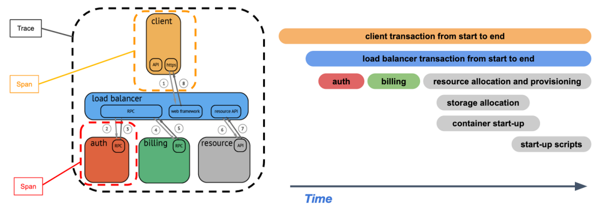

Figure 1: Spans represent individual blocks of work in a distributed system, and traces are hierarchically structured sets of spans. Figure reproduced from the OpenTelemetry documentation.

2 Tracing Distributed Systems

To profile and trace requests through distributed systems, we use the framework and tools developed by the OpenTelemetry project.[3]

2.1 Spans and Traces

To establish a causal relationship between different invocations of a method or an RPC, we record *spans*. Spans are named, timed operations that represent a piece of the execution flow in a distributed system—spans are the building blocks of traces. Each span stores a name, its start and end time, and the parent span from which it was created (if one exists). As a result, a trace is a tree of

[1]`https://github.com/codalab/codalab-worksheets`
[2]`https://github.com/rqlite/rqlite`
[3]`https://opentelemetry.io`

spans that reflects the hierarchical call stack and includes timing information. Figure 1 shows an example of spans in a trace.

To establish this hierarchical structure between spans that generate new spans, children spans must receive information about the parent span (generally the currently-executing span) when they are created and started. This is straightforward in traditional applications (e.g., with context variables), but propagating context across different services is slightly more involved.

However, in 2021, the W3C created a trace context specification that defines standard HTTP headers and a value format to propagate context information that enables distributed tracing scenarios. [4] This trace context is meant to uniquely identify individual requests in a distributed system.

As a result, to propagate this trace information across service, we simply inject the necessary trace information into the HTTP header, and extract the information when the RPC is received and the child span is created.

2.2 Instrumentation

Spans and traces provide an understandable and easy-to-use way to interpret execution information in a distributed system, but what units and functions should make up a span? This question, and the broader question of what to profile and trace, are at the core of *instrumentation*.

Instrumentation is the process of modifying an application to generate coherent spans and traces. This is generally done manually—while automatic tools for instrumentation do exist, they may be harder to interpret because they indiscriminately record all operations and/or functions. On the other hand, a carefully manually-instrumented application requires developer expertise about what units of work are salient and worth recording—instrumentation that is too high-level would generate uninformative traces, but instrumentation that is too low-level would generate traces with dozens of spans that are difficult to interpret. A balance between detail and conciseness is necessary when instrumenting, and the relevant methods to instrument are dependent on the goals of profiling and the questions to be answered. We manually instrument the applications in each of our case studies and defer details for later in the paper.

3 Case Study I: CodaLab

Background Our first tracing case study concerns CodaLab, a platform for reproducible research. Codalab `bundles` represent the code, data, and results of a sequence of commands (e.g., an experimental pipeline).

[4]`https://www.w3.org/TR/trace-context`

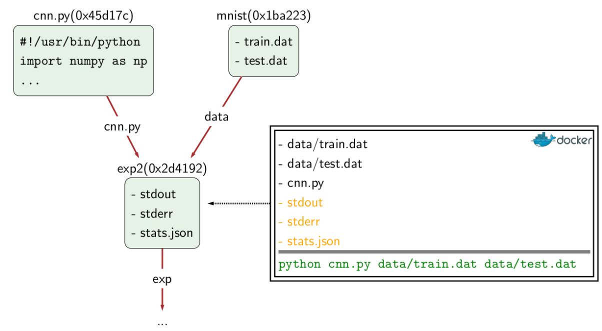

Figure 2: Each rectangle represents a bundle, and arrows represent dependencies between bundles. This figure shows two uploaded bundles. The first (top-left) is a script called `cnn.py`. The second (top-right) is a dataset (named `mnist`) with two files (`train.dat` and `test.dat`). The run bundle `exp2` is created by executing the `python` command on the contents of the `cnn.py` and `mnist` bundles. Figure reproduced from CodaLab documentation.

Users can create bundles by uploading files from their local disk. Users can also create *run* bundles, which are the output of executing bash commands on the contents of previous bundles. This shell command is executed in a Docker container by a *worker*, which may be a separate host. Figure 2 shows how run bundles are created by executing a command on data and code bundles. The dependency graph over bundles represents a reproducible path from producing the output of an experiment or set of experiments from the original code and data.

Instrumentation CodaLab is deployed as a collection of Docker container microservices (Figure 3). A single CodaLab command will involve requests that traverse between multiple of these containers, often multiple times. As a result, when observing end-to-end request latency, it can often be difficult to discern which particular service or service subroutine is causing the issue. Tracing requests as they percolate through and across services is essential for understanding request performance.

We primarily focus on instrumenting the RPC calls between services, to better understand the latency breakdown of a single slow request across different parts of CodaLab. CodaLab exposes a REST API to clients—we instrument each of these endpoints. These endpoints make further requests to a MySQL database, and we instrument each of the database queries.

Requests to study We focus our analysis on two particular types of requests—`cl run` requests and `cl search` requests.

`cl search` is the command used to return bundles that match a particular keyword or set of keywords. For ex-

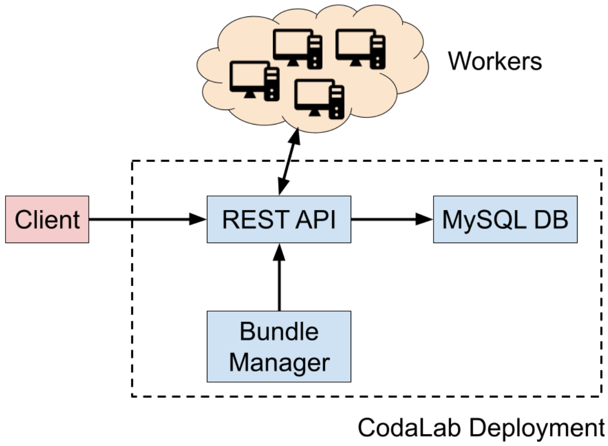

Figure 3: The architecture of a CodaLab deployment. The main three services are the REST API, the MySQL DB, and the bundle manager.

ample, `cl search owner=nfliu` would return bundles owned by user `nfliu`, and `cl search python` would return bundles with "python" in the name. CodaLab bundles store various metadata (e.g., name, owner, description, etc.), any of which can be queried with the `cl search` command. Under the hood, the command uses its input parameters to construct a SQL query, which is executed against the MySQL database containing the information about the bundles. When running many experiments, `cl search` becomes an invaluable tool for operating over sets of the experiments in batch. For example, to delete all failed experiments, one could run `cl rm $(cl search .mine state=failed -u )`. The inner `cl search .mine state=failed -u` returns a list of the bundles that are failing, and `cl rm` can take the shell output and directly operate on it. However, user experience indicates that these `cl search` queries can often be quite slow, so we trace these requests to better understand the latency breakdown across services.

Tracing Requests We run the CodaLab deployment on a AWS t3.medium instance (2 vCPUs and 4GiB of memory). We use `locust` to simulate 10 parallel clients issuing requests to the server. We measure the latency of `cl run date` and `cl search .mine` (which resolves to `cl search owner=<current user>`). As these requests are issued to the server, traces are uploaded to a different AWS t3.medium instance for post-hoc analysis and viewing. We use Jaeger to visualize the CodaLab traces.

`cl run` Trace Analysis Figure 4 presents the trace of a slow `cl run` invocation.

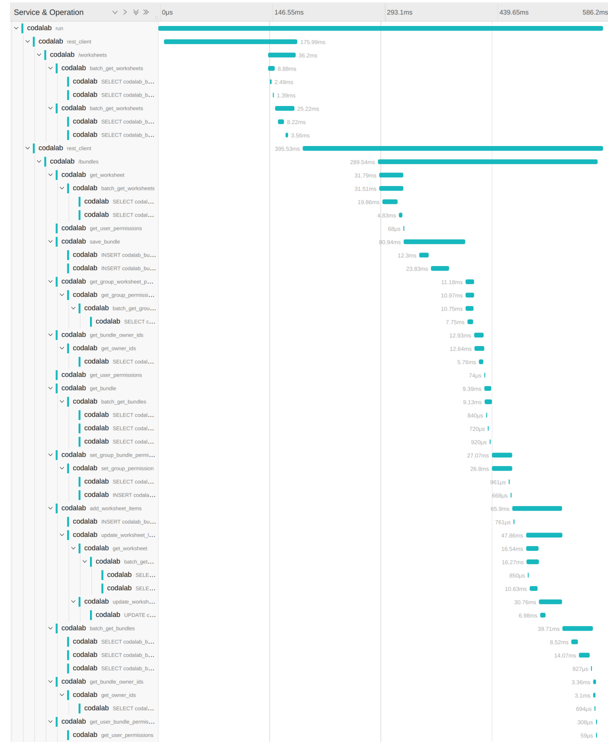

Figure 4: An example trace of a slow `cl run` invocation (586.2ms in total). The left column shows the name of the CodaLab service and operation name, and the bars on the right provide a visual timeline of execution throughout the lifetime of the request. Figure best viewed on a computer.

The command breaks up into two main REST requests. The first is a call to the `/worksheets` endpoint. When a run bundle is created, it is inserted into a worksheet, which visually hosts a collection of a bundles—the `/worksheets` endpoint returns the worksheet to insert the new run bundle to. Looking further at the breakdown of this endpoint, we see that a fairly significant amount of time is taken by network latency (i.e., the amount of time it takes for the request to be received by the server after being sent by the user). This is visually represented by the gap between the start of the `rest_client` span and the start of the `/worksheet`.

Calling the `/worksheet` endpoint itself takes 36.2ms in this example, and is mainly broken up into two calls to the `batch_get_worksheets` function. This each invocation of this function issues two requests to the MySQL database, each of which executes a `SELECT`. These requests to the MySQL database each take around half of the overall runtime of the function, and can vary in speed (from 1.39ms to 8.86 ms).

Applying a similar analysis to the other top-level REST operation in the trace (a call to the `/bundles` endpoint), we see that a significant amount of time is taken by

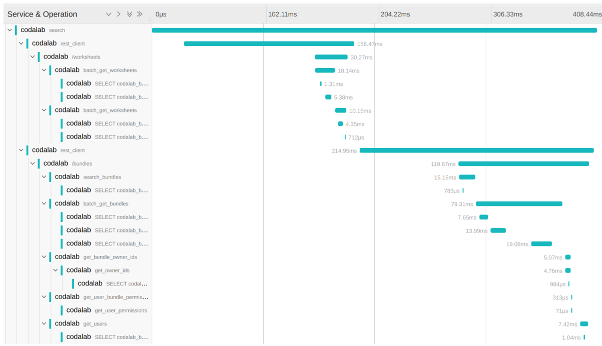

Figure 5: An example trace of a slow `cl search` invocation (408.44ms in total). The left column shows the name of the CodaLab service and operation name, and the bars on the right provide a visual timeline of execution throughout the lifetime of the request. Figure best viewed on a computer.

database operations. Furthermore, there is significant variation in the time of SQL execution depending on the supplied statement / query. For example, the shortest query takes 668 microseconds—this query inserts a new entry into the permissions database with the bundle UUID and the permissions value. The longest query takes 14.07ms—-this query selects bundles with a dependency on a supplied UUID. Studying the database schema, we find that this operation is slow primarily because it requires iterating over all database rows, since the database is only indexed by the bundle UUIDs.

As a result, our trace analysis suggests that in order to improve the performance of slow `cl run` invocations, it may be useful to focus on the performance on the MySQL database service. In particular, it may be necessary to rework the database structure to speed up frequently-executed queries. This may naturally come at the cost of storage space, since the on-disk size of the database may increase.

`cl search` trace analysis We can apply a similar analysis to slow `cl search` invocations.

Figure 5 shows the trace of a slow `cl search`. The command is again broken up into calls to two REST endpoints: `/worksheets` and `/bundles`. The `worksheets` call is exactly the same as that of the `cl run` command, so we omit the analysis for the sake of brevity. However, the call to the `/bundles` endpoint calls five subroutines, though the runtime is mainly dominated by the call to `batch_get_bundles` (79.31/119.87 ms). This function is broken up into three SQL select queries which return data about the bundles found via the search.

Corroborating our analysis of `cl run`, we see that even these simple queries can be quite slow; although they are frequently used, they require examining all rows of the MySQL database. As a result, the speed of these queries grows linearly with the number of bundles on the server. Given that CodaLab servers can easily host tens of thousands of bundles, this could explain why users experience significant (often longer than 60 seconds) latency when using the production deployment of CodaLab. For better or for worse, it appears that the database service is the culprit for both slow `cl run` operations and slow `cl search` operations, so our recommendations of refactoring the schema and adding additional database indices for commonly-executed queries also apply here.

4 Case Study II: `rqlite`

For our second case study, we analyze the traces from `rqlite`, a replicated SQLite system written in Go. `rqlite` uses Raft behind-the-scenes as its consensus protocol. In conducting this case study, our main goal was to get a better sense of the actual performance costs associated with replication (and Raft in particular), in terms of both individual Raft RPC performance and end-to-end-request performance.

Instrumentation We instrument the Raft `AppendEntries` and `RequestVote` RPCs, since we wanted to get a sense of (1) how often these RPCs were executed and (2) the individual cost of each RPC. We also applied some instrumentation to the parent functions that called these RPCs (e.g., the `heartbeat` function), to get a better sense of why these RPCs are executed.

Experimental Setup We run `rqlite` nodes on AWS t3.medium instances (2 vCPUs and 4 GiB of memory). To see how end-to-end request latency on the replicated database changes as more nodes are added (and more communication overhead is thus incurred), we experiment with varying cluster sizes: single-node (no replication), and clusters of 3, 5, 7, and 9 nodes. `rqlite` exposes a REST HTTP API to the replicated database, and we use `locust` to simulate 10 parallel clients issuing requests to the replicated database. In particular, we the workload is evenly divided into database writes and reads.

Our test table contains contains two fields: `name`, of type `TEXT`, and `age`, of type INTEGER. For the database writes in our workload, we create a new entry with a random name and random age and insert it into the database. For the database reads in the workload, we `SELECT` all entries fro m the test table with a randomly-chosen name. The test database is stored in memory, since we are primarily interested in the speed and cost of inter-node communication, rather than any potential costs associated with

	Write Requests (ms)		Read Requests (ms)	
Cluster Size	50%	95%	50%	95%
1	42	53	41	50
3	45	53	41	50
5	44	56	42	50
7	49	58	44	52
9	50	59	45	52

Table 1: End-to-end request time for read and write requests for varying cluster sizes.

	AppendEntries (μs)		RequestVote (μs)	
Cluster Size	50%	95%	50%	95%
3	132	198	286	379
5	157	182	238	398
7	143	173	358	408
9	135	191	281	351

Table 2: RPC latency for varying cluster sizes. We omit cluster size 1 because no RPCs are sent in this setting.

disk access.

Results: End-to-End Request Latency We first benchmark end-to-end request latency with our previously-described workload across a range of cluster sizes. Table 1 displays the median request time, as well as the 95th percentile of request times, for the read and write operations for various cluster sizes. Even in our small-scale benchmark, we can observe both median and 95th-percentile response times increasing slightly—the performance overhead from replication is slightly noticeable.

Results: Individual RPC performance To better understand how individual RPC performance contributes to overall request latency, we study the cost of the `AppendEntries` and `RequestVote` Raft RPCs; Table 2 displays the results. The cost of each RPC is on the order of hundreds of microseconds. Furthermore, as expected, the cost of an individual RPC does not significantly change with the cluster size. Rather, as clusters grow or shrink in size, the total number of RPCs required to commit an operation increases.

The number of recorded `AppendEntries` RPCs far outpaces the number of `RequestVote` RPCs, since the former is frequently invoked as a heartbeat from the leader to each of the followers, while the latter is only really used when candidates must gather votes to elect a new leader.

Overhead from instrumentation We also run some preliminary experiments to assess the overhead caused by instrumentation itself. In particular, we reran the same experiments with an unmodified (and thus uninstrumented) distribution of `rqlite`. However, we found that we were not able to discern a significant different in end-to-end request latency nor single-RPC latency between these two versions in our simple testbed. Although it is entirely possible that heavier loads or other experimental settings would reveal a significant overhead to tracing, we did not observe any differences.

Discussion By analyzing the performance of both end-to-end requests and the individual Raft RPCs, we gained a better understanding of the overhead associated with replication and Raft in particular. When reading the abstract description of an algorithm, it's often unclear which operations could be potential bottlenecks, or the relative frequency with which they're invoked—tracing these requests allowed us to concretely measure how long a given RPC takes (not too long, as expected), and how often it's called.

Furthermore, while it's natural that replication will increase end-to-end performance, it can be unclear how this affects request times in benchmarks. Although this benchmark is simple, we were able to observe a slight decrease in performance as the result of replication, making these potential performance hits much more concrete. In our setting, replication comes at a relatively small cost, though it may cross the threshold for intolerably expensive in other applications that see higher load or larger operations.

5 Conclusion

In this work, we traced and profiled two different distributed systems: CodaLab, an application composed of several microservices, and `rqlite`, a distributed SQLite database. Our case study of CodaLab helped us better understand application hotspots and the complete life-cycle of a request, pointing to potential areas for performance improvements. Our study of `rqlite` improved our conceptual understanding of the Raft consensus algorithm by making its operation concrete and helped us get a hands-on sense of the order of magnitude of performance hits associated with replication. More broadly, we adapted the same set of tracing tools to answer very different questions about different distributed systems, providing a template for instrumentation and other design decisions when conducting future analyses.

References

[1] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.