RPCBOARD:
Reconstructing and Visualizing Distributed Traces from Log Data

Garrick Fernandez
garrick@cs.stanford.edu
Stanford University
Stanford, CA

Abstract

Remote procedure calls (RPCs) are a foundational building block for distributed systems. RPC libraries allow developers to write a language-agnostic description of an RPC service and compile client stubs and server code that allow computers to communicate with one another. With multiple nodes working in concert to present the abstraction of a unified service, the tasks of debugging, analyzing, or simply understanding distributed systems become harder. We focus on the goal of making a distributed system observable, developing an end-to-end framework, called RPCBOARD,[1] *that collects log data from server and client interactions, reconstructs a timestamped call graph, and visualizes playback of the call graph in real time. We consider tradeoffs between performance, accuracy, detail, and the overhead of instrumentation and code modifications needed to reconstruct these traces.*

1. Introduction

When developing distributed applications that utilize RPC's, it can be useful to see how the application is operating as a whole. To this end, we present RPCBOARD, a framework that instruments and visualizes the operation of distributed applications on the C/C++ gRPC core. We hope this tool would be useful to students or developers looking to gain intuition for how their application is working.

In section 2, we further define what an RPC is, as well as introduce gRPC, a popular library for implementing distributed applications. In section 3, we discuss the goals we want to achieve and simplifying assumptions we made over the course of the project. In section 4, we discuss the architecture of the system in depth, outlining each of RPCBOARD's three major components and design problems that were solved for each. In section 5, we evaluate the framework. In section 6, we discuss related works that contributed to the formation of the project.

2. Background

In a remote procedure call (RPC), a procedure is executed in a separate address space from the caller (typically, on another machine, over a network). Clients do not have to deal with the specifics of establishing communication between these address spaces, as well as marshalling procedure parameters and data to and from a serializable form that can be sent over the wire.

An RPC library helps developers write an RPC service; one such library is gRPC[1], open sourced by Google in 2015. gRPC uses Protobufs as its interface definition language—developers describe the RPC service in a `.proto` file, and use the `protoc` compiler to produce language-specific stubs and server code. Communications are typically sent over HTTP/2.

gRPC is popular because it has libraries in multiple languages, including Python, C++, and Go. The libraries for all C type languages make API calls to the same C/C++ core (Figure 1).

3. Goals and Assumptions

Our goals are to create an instrumentation and visualization framework that takes little effort to get set up, and offers users a good intuition of how their system

[1] The code is available at github.com/garrickf/rpcboard.

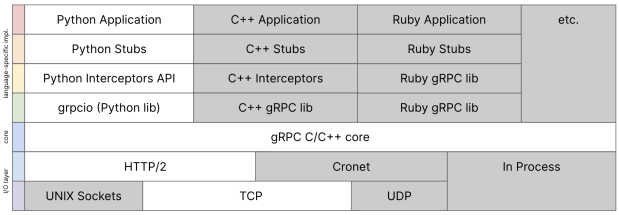

Figure 1. The gRPC language stack. The languages we focus on are highlighted in white.

is running. We were inspired by previous visualizations and work done on visualization frameworks (see Related Work), so we wanted to create something that could collect data for a distributed trace in real time and play back a visualization of it. We proceeded in two milestones:

1. Investigate gRPC and produce a method for generally instrumenting applications and assembling a distributed trace. An idea we borrow from profiler tools such as `gprof` is the idea of a **hierarchical call graph**—who called what, and when. This provides a useful utility for developers in of itself, and it's a stepping stone to visualizing a distributed trace.

2. Take the hierarchical call graph and turn it into an animated visualization. This was the focus on the front end, described more in the next section.

We developed on a single machine and, for the sake of simplicity, focused first on developing traces that worked under ideal conditions (no dropped connections, no clock skew).

4. Architecture

The overall architecture consists of three components. On the RPC side, we prompt clients and servers to produce HTTP/2 trace log data, which we forward to a central log server. This log server, the second component, takes log data and reconstructs a distributed trace with it. The third component is a dashboard that can visualize and play back distributed traces. The components were developed primarily in JavaScript (Node.js) and Python, with additional reading of the gRPC core source code, which is in C++.

In the next three sections, we flesh out each of these components in more detail, describing design decisions and interesting challenges we had to solve.

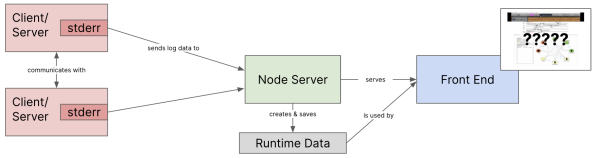

Figure 2. The RPCBOARD architecture.

4.1. RPC Layer

At the RPC layer, we present a script, `rpcboard-trace`, that is meant to be called with the executable that starts servers or client calls in the target distributed application. This script makes a streaming HTTP/TCP connection to the log server (which is started with a call to the `rpcboard` script), and configures the running process to forward all of its log data to the log server for processing.

Why send log data? There are indeed other places where instrumentation can be inserted into an RPC service. For example, one could write additional stubs that pass around trace information, and require the application developer to use those. Further, one could use the gRPC specific interceptor API, which allows developers to access the parameters and context of an RPC specifically at the moments clients and server send and receive them. At the network layer, if one routes all RPC traffic through a web of proxies under our control, we have access to all communication, which would also allow us to profile the system. The issues with these instrumentation strategies, and why we chose to go with log data, was that the aforementioned approaches require additional overhead, either in terms of code or infrastructure. In the interest of creating a framework that was easy to plug in and use, it was appealing to make it rely on logging code that already exists within gRPC. It is also an interesting problem whether a distributed trace can be faithfully recreated from log data, a problem we further discuss in the next section.

One large challenge at this layer was the determination of what data was salient to send, in order to produce an accurate, detailed distributed trace. gRPC offers several trace flags that allow one to log everything from API calls to the C++/C core, to timers in the gRPC internals, to load balancing and DNS resolution. For our traces, we found that the `http` flag, which traced state in the HTTP/2 transport engine (in

other words, at a layer below the core), provided the information we wanted.

The reason why this provided sufficient information to recreate a trace is due to the way gRPC handles transport operations. In the gRPC core, collections of steam operations (a struct called `grpc_transport_stream_op_batch`) are sent to transports, which represent the ongoing communication between a client and server[5]. These transport operations are processed asynchronously, and their results are returned via a completion queue, which may trigger additional logging. The gRPC core prefixes each log line with a timestamp (resolution in nanoseconds), a thread id (TID) of the thread prompting the logging, and the source file and line of the code that triggered the log. Log lines involving transport operations also include the address of the transport (transport ID). Combining the transport operations with their associated metadata allows us to reconstruct a distributed trace.

To illustrate, a sample timeline of stream operations for a client sending a request to a server could look like:

1. A client sends initial metadata.
2. A server receives initial metadata.
3. A client sends a message.
4. A client sends trailing metadata.
5. A server receives a message.
6. A server receives trailing metadata.

4.2. Back end layer

The back end log server is initialized with a call to `rpcboard`. On one port, the server listens for connections from rpcboard-trace processes; on another port, the server serves the front end dashboard, presented in the next section.

The server's main task is to ingest incoming log information and construct it into a hierarchical call graph. To accomplish this, we developed a solution where larger and larger primitives are constructed with observational data from the log. At each step, we use the currently collected data to cross-reference and match events until we can construct the full hierarchical call graph.

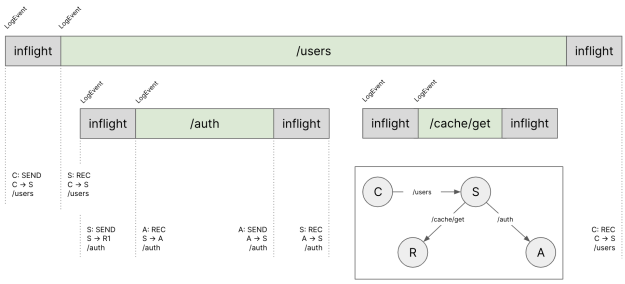

Figure 3. Diagram of an example distributed trace. One RPC call spawns two additional RPC calls to other services. A sequence of (client send, server receive, server reply, and client receive reply) forms a Span.

At the first layer, transport stream operations are arranged into observations called **LogEvents**. Each LogEvent contains one server's perspective of a chain of communication: either they were the sever receiving a request and processing it, or they were a client sending off a request and waiting for the reply.

These log event objects are created and emitted by a **Parser** to another class, called a **Spanner**. The Spanner combines LogEvents into **Spans**, which contain both perspectives of the RPC communication (Figure 3). Authorities, and paths, are used to cross reference and assemble Spans.

Spans are then structured into a tree-like object, where parent nodes are responsible for making the RPC calls below them. Thread ID's and transport ID's are used to cross reference Spans here. The **TreeGenerator** class is responsible for this reconstruction and also for keeping track of node metadata (such as authority names, edges, and unique node ID's for the front end).

4.3. Front end layer

The front end is a dashboard that allows users to visualize the execution of a distributed application. It queries the server for new, compiled distributed trace information at set intervals, then visualizes the ingested data using React and three.js, a Javascript 3D library with a default WebGL renderer.

To make it easier to understand what is happening in a distributed trace, a canvas is presented where nodes are drawn as circles, and "packets" containing RPC information are animated moving between them. Edges

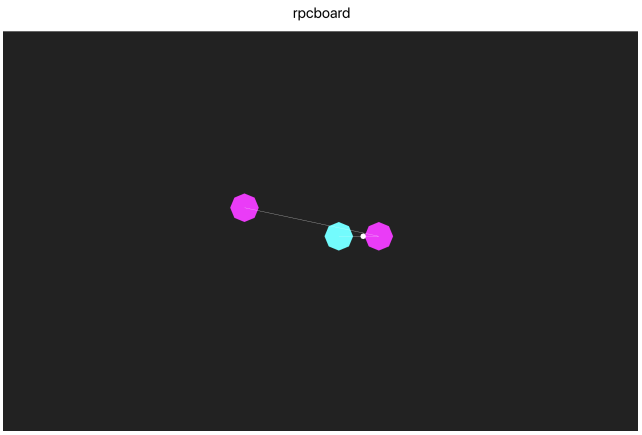

Figure 4. The working dashboard prototype, with nodes and animated RPC messages. Further work would add a playback bar and timestamped graph.

are drawn between nodes that communicate at least once over the entire trace, giving a view of the total architecture of the system.[2].

Once a hierarchical call graph is established in the back end layer, the matter of drawing the graph on the canvas is a different problem. Unlike the structure of the call graph, which shows causality, we want a way to query relevant animation information at given points in time. Additionally, we would want to be able to query for relevant events across a range of times, so we can display relevant information to where the user currently is (such as a playback or scrub bar). All the time, we want these to be done with a small number of operations, for performance.

These lend well to a B+ tree, which supports fast point queries, due to the high degree of its nodes, and fast range queries, due to the leaf nodes, which store values connected in a linked list. We developed a modified B+ tree structure called an `AnimationQueryTree` in order to translate and store the distributed trace.

Each of the internal, or index, nodes of the tree have $n + 1$ keys and n values. The additional key allows each internal node to know the range of time it has information about. When the tree is constructed, all timestamps are extracted from the hierarchical tree and sorted; the ranges between each timestamp become a leaf node in the `AnimationQueryTree`. Like the index nodes, each leaf node also knows it's start and end ranges, so an `AnimationQueryTree` can be built from the leaves up to the root, with each node's start and end ranges forming the keys in the parent node.

When an `AnimationQueryTree` is being filled with information, the leaf node corresponding to the start of the event is found. An `AnimationInfo` object is appended to a list of objects in the leaf node, and all leaf nodes following it until the end time of the event is reached. During animation, the current timestamp is used to find the relevant leaf node, which contains all events that need to be represented or animated on the front end. The leaf node's start and end ranges, as well as the current time in the animation, can be used to interpolate element postions on the canvas.

5. Evaluation

```
children:
    span:
        client (TID 4549262784) sends RPC /pingpong.PingPong/askEvenOrOdd at [2020.6.11]0:57.29.868371000
        destination node: localhost:50000
        server (TID 123145398517760) recvs at [2020.6.11]0:57.29.869108000
        server (TID 123145398517760) sends reply at [2020.6.11]0:57.29.876829000
        client (TID 4549262784) recvs reply at [2020.6.11]0:57.29.877022000
    children:
        span:
            client (TID 123145398517760) sends RPC /pingpong.PingPong/askEvenOrOdd at [2020.6.11]0:57.29.871932000
            destination node: localhost:50001
            server (TID 123145416454144) recvs at [2020.6.11]0:57.29.872619000
            server (TID 123145416454144) sends reply at [2020.6.11]0:57.29.876389000
            client (TID 123145398517760) recvs reply at [2020.6.11]0:57.29.876605000
        children:
            span:
                client (TID 123145416454144) sends RPC /pingpong.PingPong/askEvenOrOdd at [2020.6.11]0:57.29.875345000
                destination node: localhost:50000
                server (TID 123145415307264) recvs at [2020.6.11]0:57.29.875799000
                server (TID 123145415307264) sends reply at [2020.6.11]0:57.29.875947000
                client (TID 123145416454144) recvs reply at [2020.6.11]0:57.29.876141000
            children: (none)
```

Figure 5. Pretty-printed hierarchical call graph. Each node has an associated Span (except for the root node), and a number of children nodes that represent calls that the parent made.

The system was evaluated on two sample RPC applications we developed using the Python localization of gRPC. One application, `pingpong` (seen in the demo and Figure 5), consisted of two servers passing a number back and forth, decrementing it by two each time. When the number reached 0 or -1, that server responded with whether or not the number was even, and the messaged bubbled back up to the client who sent the original number.

Since communications are animated, relative latency can be observed among communications, which allows for some level of profiling a distributed application.

[2] A video of a working draft of the visualizaion (not shown in the original demo and presentation) can be found here: youtu.be/5aYhiM84PqU.

6. Related Work

RPCBOARD is inspired by and builds upon previous work on observability and distributed tracing.

In the RPC layer, alternative instrumentation approaches, such as OpenTelemetry and Istio, intercept RPC communication at different layers of the language stack (the interceptors API and network, respectively). RPCBOARD opts for gRPC logging information, as this comes out of the box.

The idea of using application logs to extract performance insights is the driving idea behind Splunk, which allows customers to sift through and visualize large amounts of heterogeneous log data. RPCBOARD takes a slightly more opinionated approach, throwing irrelevant log information away during parsing, and surfacing only the hierarchical call graph for visualization on the front end.

OpenTelemetry[4], a nascent project and the result of a 2019 merger between existing projects OpenCensus and OpenTracing, seeks to create robust, portable telemetry via a unified API and SDK that's agnostic of the backend collecting and processing information and the front end ingesting it. RPCBOARD borrows and adapts the specification of a "span" in producing its own trace primitives. However, whereas OpenTelemetry (on gRPC) relies on the interceptor API to insert tracing information into each RPC call for collection, the log server reconstructs the timeline of execution by corroborating log data after collection.

Netflix's project Vizceral[2] heavily inspired the front end design, informing the tech stack (React and three.js) and the general procedure of ingesting data and drawing it. In contract to Vizceral, the graph data in RPCBOARD is more complicated, since we produce an `AnimationQueryTree` that can trace the flow of individual RPC calls in a trace. Vizceral opts for a more high-level picture, assigning an aggregate measure of network traffic to each edge in the graph, and drawing a "flow" of dots to simulate, but not reproduce, the requests being sent.

Diego Ongaro's Raft Scope visualization[3] also served as visual inspiration, with its use of color coded messages and graphics encoding node and system state.

TensorBoard, a framework for visualizing machine learning experiments in TensorFlow or Pytorch, was the initial inspiration for the idea of an end-to-end instumentation and dashboard solution. RPCBOARD follows a similar process of collecting "summary operations" from the run of an experiment, and serializing that information into files for access on the front end.

7. Conclusion

In this paper, we present RPCBOARD, a minimally invasive instrumentation and visualization framework that works with gRPC applications on the C/C++ core. In developing this, we contribute the following ideas:

1. A taxonomy of vital information that must be provided (or produced) through application logs in order to reconstruct distributed traces.

2. Primitives for trace reconstruction that allow us to build the distributed equivalent of a hierarchical call graph.

3. A B+ tree-like lookup structure that can be constructed from the call graph to efficiently determine which elements need to be drawn and animated.

While many of the front end components remain yet to be fleshed out, the server is able to produce distributed traces in a hierarchical tree format, and these trees can be translated into `AnimationQueryTrees` for use on the front end. Since the pathways for information transmission and transformation are in place, future development on this framework could focus on utilizing more information stored in the `AnimationQueryTree` to further improve the visualization and add supporting metadata to the dashboard, such as the authorities, RPC call info, and more.

8. Acknowledgements

Special thanks to David Mazières and the rest of the CS244B course staff for their advice and guidance throughout the project.

References

[1] gRPC. grpc.io. 1

[2] Netflix. Vizceral open source. 5

[3] D. Ongaro. Raft scope. 5

[4] OpenTelemetry. opentelemetry.io. 5
[5] V. Pai. Transport explainer. 3