Request Level Fault Injection

Tuan Tran, Peter Alvaro

University of California, Santa Cruz
{atran18, palvaro}@ucsc.edu

Abstract

Recent years manifest the growing presence of microservices as the dominating architecture for the modern landscape user-facing applications. Microservice architectures (or simply "microservices") are characterized as a thoroughly decentralized, distributed topology of services providers designed to coordinate in response to client requests. Accordingly, given the constant specters of asynchrony and partial failure, the occurrence of potentially devastating fault scenarios is not a question of 'if' but 'when'. In this paper, we present Request Level Fault Injection (RLFI) as an efficient approach to performing fault injection experiments on large-scale microservice architectures which minimizes disruptions on the critical path. We present a framework allowing developers to trigger faults on microservices at the request level and obtain fine-grained traces of the system for analysis. This work builds upon a popular tracing infrastructure known as OpenTracing [3] to support data propagation over the wire while providing complementary traces of the system in response to fault injection (FI) experiments. We describe our motivation for the work, the benefits of adoption, a proof-of-concept implementation, and an evaluation of the framework prior to speculating about a number of future directions.

1 Introduction

In a world characterized by increasingly powerful and increasingly distributed applications, the cost of failure translates very directly into devastating repercussions. Violations in software correctness could cost Facebook millions in ad revenue by the hour, Amazon tens of millions by the hour if the day happens to be Black Friday, or cause an unfortunate administrator to have to wake up in the middle of the night to fix the problem [[need real examples with citations; (extra attention-grabbing boost) give example of devastating non-monetary costs. Has anyone died because of distributed system failures? – kmd]]. Accordingly, the benefits of thorough testing translate into very valuable defenses protecting large quantities of revenue against the hands of fate. Companies typically invest in active testing techniques in the form of a fault injection framework as a preferred method for challenging the correctness guarantees of distributed services and, in the process, preempt the manifestation of potentially devastating software errors. However, modern fault injection frameworks are not intelligent enough to maximally identify a set of existing failure scenarios within the time constraints of a production environment. Tools like Netflix's ChaosMonkey [7] are unlikely to find certain errors caused by intricate combinations of failures [8] and suffer unknown soundness and completeness guarantees due to a crippling reliance upon randomness. Additionally, fault injection frameworks such as Orchestra [10] require developers divert the time and energy from forward-driving design and implementation problems toward the inconvenient and tedious task of application-level instrumentation.

An ideal fault injection system design allows the careful targeting of individual components for particular failures, requires little work to integrate into the overall service, and outputs traces of the whole system as a result of individual experiments. To realize these goals, we propose *Request Level Fault Injection* as a tracing system with an integrated fault injection component to provide both fine-grained fault injection and traces.

2 Background and Motivation

This section will provide an overview of Opentracing, the types of wire protocols that we explore for cross-process communication and how annotations are propagated through them, and the reasoning behind integrating the fault injection mechanism into the tracing framework. [[need a proper intro – kmd]]

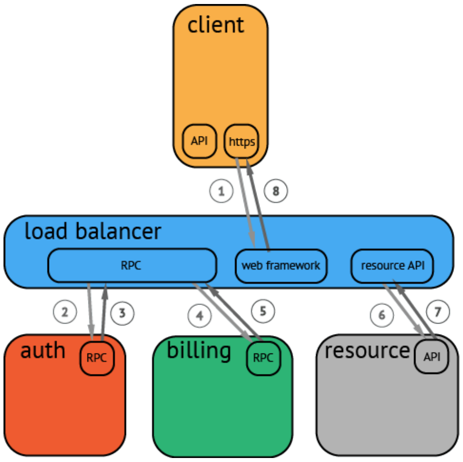

Figure 1: A sample trace

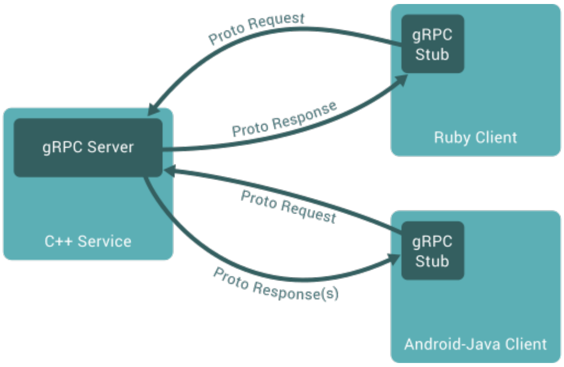

Figure 2: Architecture of gRPC

2.1 OpenTracing

Today's microservice architectures are large in scale and highly complex and often consist of services written many languages. The characteristics render the process of developing a principled and useful distributed tracing framework over these services extremely challenging. Opentracing [3] provides a set of vendor-neutral APIs for obtaining distributed traces through annotation propagation and allows for easy integration into existing microservice architectures. Previously available solutions for tracing through application level annotations such as Magpie [9] and X-Trace [11] are unappealing because of the per-application schema requirement exacerbates the level of integration inconvenience and introduces higher application overheads [13]. For these reasons, we chose to use the Opentracing APIs to provide our architecture with fine-grained traces.

2.1.1 Trace Terminologies

A **span** is a basic, logical unit of work with a start time and duration and exists within a single service. Spans can be nested to show causal relation. **Baggage** is a set of <K,V>pairs wrapped within a span's context and allows for transparent propagation of arbitrary data. A **tracer** consists of a set of spans belonging to a group of services. For example, a set of REST endpoints belonging to a group of services in a server manifests a tracer. For cross-process tracing, a tracer will **inject** the span's context, and **extract** it on the other end. [[definitions of inject/extract? – kmd]]

2.1.2 Trace flow

Each service that a developer wishes to be traced will need to have a span constructed, after which it will be added to a list of spans in the global tracer. Each span will automatically provide trace detail consisting of a timestamp and duration; the developer can decorate the span with more details, such as the status of a lock or value of a variable by adding to the span's baggage. When a service calls another service, the tracer will have to be invoked to inject the local span's context into the wire, and extracted on the other side. During the extraction, a new(child) span is constructed from old(parent) span's context. Spans are recorded in the order that they are finished, meaning that whichever finishes first will be first to output trace information to the output stream. Figure 1 shows a simple trace involving several microservices [3].

2.2 Fault Handling

There are three ways one could approach adding a fault injection component into a microservice architecture. The first approach consists of incorporating the FI component into the constituent services of the microservice architecture individually. Each service will have a special chunk of code capable of reacting to fault injection information received from upstream. [[what kind of reactions? – kmd]] The obvious advantage of the approach is the ease of implementation. Additionally, the developer exerts full control on how each service should react to a type of failure. However, the approach requires the existence of FI code in all target services. Such a task is intractable if particular microservice architectures contain hundreds of services written in a multitude of languages [[example(s)? – kmd]].

The second approach places the onus of fault handling upon the wire protocol. If all services communicate through a single protocol (e.g. HTTP), then the method greatly reduces the absolute amount of code required to support and manage fault triggers. Accordingly, even if subsets of services use different protocols, the amount of code devoted to FI support is proportional to the number of different communication protocols, which should be in the single digits. The downside of the approach is the need to extend the bare communication protocol, which may necessitate modifying hardened core components in a process very vulnerable to the possibility of introducing new bugs capable of violating the integrity of the communication protocol.

The third approach, which represents the core philosophy behind RLFI, advocates the incorporation of fault handling mechanisms at the programming language level. [[what do you mean by programming language level? need a paragraph description. – kmd]] The method is practical from an implementation perspective because the number of languages in a service is bounded between ten and twenty [[citation? – kmd]]. The approach also avoids the risk of polluting both the underlying application code and the code for the chosen wire protocol(s). As an example, to incorporate a FI framework over a set of HTTP REST endpoints written in golang [1], the only necessary changes are wrapping the communication protocol handler functions inside decorators. Below is a code listing of how one would implement this: [[make this a figure – kmd]]

```
func homeHandler(w http.ResponseWriter, r *http.Request) {
    //stuff goes here
}

func decorate(f http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        //do some preprocessing of the request
        f(w, r) //call the function
    }
}

func main() {
    http.HandleFunc(”/home”, decorate(homeHandler))
    http.ListenAndServe(”localhost:8080”, nil)
}
```

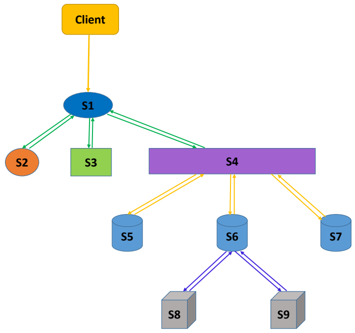

Figure 3: Test architecture consisting of a small set of microservices

Notice that the changes to the application are minor and the decorator has full access to whatever was passed over the wire.

2.3 Wire Protocols

Wire protocols are used for writing application-level code for cross-process communication. Protocols such as HTTP, gRPC [2], and Thrift [5] support the sending and receipt of data between services within mircroservice architectures. Opentracing supports both HTTP and gRPC as wire protocols and uses both to support the propagation of span context between particular, but arbitrary, services throughout the system. In gRPC, clients expose methods called remotely from servers to support client call management. Data passes along gRPC wires in the form of Protocol Buffers [4]. Figure 2 illustrates the basic flow of the gRPC architecture [2].

2.4 Motivation

We believe fine-grained traces and request-level fault injection are complementary. One of the primary motivations for obtaining traces of large-scale distributed services is to support analyzes of results derived from bad executions to identify weak points capable of preventing appropriate responses to client requests. Such weak points represent ideal targets for fault injections in experiments verifying failure scenarios. We would like to take advantage of Opentracing's built-in mechanisms for easily passing arbitrary baggage between services to propagate request-level faults through baggage items.

3 Implementation

[[needs a revamp. what are the architectural components of RLFI specifically? What is the process of slapping

RLFI over an existing microservice architecture? A diagram would be awesome. – kmd]]
A number of implementation choices influenced the construction of the RLFI framework. Because we do not have a real microservice architecture to perform tests on, we built a simple service consisting of two **servers** listening on two different ports and a set of services contained in each server, the architecture of which can be seen in Figure 3. The services are coded to construct spans upon being called, and will provide detailed traces of their states. The **client** will start by invoking service1 on the first server, which will trigger the full call graph and provide the complete trace.

We chose **golang** [1] as our programming language because of its extensive support for the Opentracing framework and because of the strength of its **decorator** syntax. In total, the lines of code is just shy of 500 lines including the code for the servers, client, and the fault handling mechanism. The wire protocol chosen was **HTTP** in the form of REST method calls, and support provided by Golang made it very easy to integrate a decorator that wraps around all of the services.

We leverage the powerful **Baggage** annotation provided by Opentracing to propagate our failures downstream. The client can accept as argument any service that it wants to target for failure testing, and will inject a baggage item whose key serves as a flag into its own span. This baggage is then propagated to whichever service that it calls, and further downstream if the said service calls upon other services all of which can see the client span's baggage items. Because we have decorated our handler to intercept service calls, our decorator will detect if the failure flag exists before proceeding to calling the service itself. If the flag exists, and the decorator notices that the service at hand matches that which has been signaled for failure, then it will act and perform the injection.

Two of the most common types of failures in distributed services are *unknown delays* and *packet lost*. When services send data to one another, we can never be certain when, or if, the data will arrive. In our implementation, we allow the tester to specify the delay time to see how the upstream services react, and the decorator will simply sleep for this duration before calling the service itself. To simulate packet lost, we have the decorator ignore the request and never actually call the service.

The verification step to see if our fault injection does work as intended is fairly straightforward. To verify that a delay is actually injected, we simply check that a service's span traces will be delayed for said period of time before being printed. To verify that a packet has indeed been dropped, we check that the service's span traces is not printed along with the other spans' traces.

4 Clade (working name)

//An example microservice architecture

4.1 Service 1

//

4.2 Service 2

//

4.3 Service 3

//

4.4 Service N

//

5 Evaluation

//

6 Related Work

Previous work on fault injection such as the Netflix Simian Army [7] provides a set of tools for inducing faults such as randomly crashing processes have been released on their webservice running on Amazon's cloud. The downside to randomly crashing nodes is the cost of restarting the nodes again, which RLFI avoids. Orchestra [10], a fault injection environment developed by Scott Dawson et al. [[for what purpose? – kmd]] requires changes to the raw socket API, which might seem appealing to those who prefer application level instrumentation. Ferrari [12] is similar to RLFI with respect to the reliance on software traps triggered by events such as memory access to actually inject the faults. [[need to mention gremlin – kmd]]

Magpie [9] is a modeling service capable of collecting request-level traces across a distributed system. However, the approach requires applications follow a specific schema [[need more detail – kmd]]. X-trace [11] also provides fine-grained traces through an annotation propagation scheme, but the performance is greatly impacted because of the abundance in metadata recorded. Zipkin [6] requires application-level implementation of the client and server and contains components not useful for our aim [[such as... – kmd]]. Dapper [13] greatly resembles the Opentracing architecture, but it is internally deployed at Google.

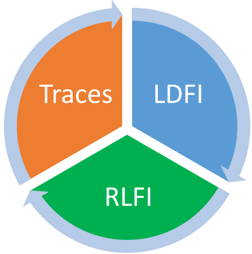

Figure 4: Fine-grained tracing, LDFI and RLFI completing the puzzle of debugging distributed systems

7 Future Work

[[need more future directions. – kmd]] The current implementation of RLFI is a proof-of-concept to show the utility and efficiency of the approach to fault injection experiments. Our current focus is extending the implementation to support a multitude of languages, not just Golang.

It is next to impossible to obtain a real life service from industry partners to perform our tests on, mainly because our partners do not want to risk compromising service integrity or jeopardizing confidential information such as software architecture and client data. Also, when compared to the real world, the toy architectures examined in the evaluation are practically trivial.

Manually digging through traces to find interesting points of failure for RLFI is a daunting task to say the least. PyLDFI, an implementation of Lineage Driven Fault Injection, is pioneered by researchers at UC Santa Cruz [[need technical report on pyLDFI to cite here – kmd]] capable of pinpointing failure scenarios ideal for FI experiments driven by RLFI. We can feed the good traces produced by the underlying into LDFI, which will find identify critical points of failure. With the critical points found, we carefully inject the faults into the system and determine whether the fault handling mechanism of the service is sufficient. We continue such a cycle of tracing, reasoning, and injecting failures until we identify enough devastating bugs to preoccupy the lives of project systems builders or, in the absence of bugs, are thoroughly convinced of the correctness of the implementation.

Currently the traces resulting from our framework are cluttered and require parsing to support legibility. We would like to develop a small, offline tool capable of transforming the traces into something PyLDFI can understand and also produce a visual call graph for ease of verification.

8 Conclusions

Intelligent failure testing in modern microservice architectures plays an important part in preventing rare bugs from causing headaches to developers and keeping services available to clients. Previous approaches to failure injection are either missing important combinations of failure due to the randomness involved, or require modifications to the low-level communication interface. RLFI works by propagating failure flags through *baggage* annotations provided by Opentracing [3], and handle faults by wrapping a decorator around the wire protocol's handler function as described in section 3.

In addition to fault injection at the request level, RLFI is integrated with a tracing framework that will provide fine-grained traces of the underlying system both before and after the injection experiment. Through Opentracing, we can piggyback failure flags over the wire, and visually verify system response from the resulting traces. RLFI with tracing and LDFI [8] allows us to complete the puzzle of debugging distributed systems.

References

[1] Golang. https://golang.org/.

[2] Grpc. http://www.grpc.io/.

[3] Opentracing. http://opentracing.io/documentation/.

[4] Protocol buffers. https://developers.google.com/protocol-buffers/.

[5] Thrift. https://thrift.apache.org/.

[6] Zipkin. http://zipkin.io/.

[7] The netflix simian army. http://techblog.netflix.com/2011/07/netflix-simian-army.html.

[8] ALVARO, P., ROSEN, J., AND HELLERSTEIN, J. M. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 331–346.

[9] BARHAM, P., ISAACS, R., AND NARAYANAN, D. Magpie: online modelling and performance-aware systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS-IX)* (May 2003).

[10] DAWSON, S., JAHANIAN, F., AND MITTON, T. Orchestra: A fault injection environment for distributed systems. Tech. rep., In 26th International Symposium on Fault-Tolerant Computing (FTCS, 1996.

[11] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In Proceedings of USENIX NSDI, 2007.

[12] Kanawati, G. A., Kanawati, N. A., and Abraham, J. A. Ferrari: A flexible software-based fault and error injection system. *IEEE Trans. Comput. 44*, 2 (Feb. 1995), 248–260.

[13] Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. *Google Technical Report* (April 2010). www.hello.com.