3MileBeach: A Tracer with Teeth

Jun Zhang
UC Santa Cruz
Santa Cruz, California
jzhan293@ucsc.edu

Robert Ferydouni
UC Santa Cruz
Santa Cruz, California
rferydou@ucsc.edu

Aldrin Montana
UC Santa Cruz
Santa Cruz, California
akmontan@ucsc.edu

Daniel Bittman
UC Santa Cruz
Santa Cruz, California
dbittman@ucsc.edu

Peter Alvaro
UC Santa Cruz
Santa Cruz, California
palvaro@ucsc.edu

ABSTRACT

We present 3MileBeach, a tracing and fault injection platform designed for microservice-based architectures. 3MileBeach interposes on the message serialization libraries that are ubiquitous in this environment, avoiding the application code instrumentation that tracing and fault injection infrastructures typically require. 3MileBeach provides message-level distributed tracing at less than 50% of the overhead of the state-of-the-art tracing frameworks, and fault injection that allows higher precision experiments than existing solutions. We measure the overhead of 3MileBeach as a tracer and its efficacy as a fault injector. We qualitatively measure its promise as a platform for tuning and debugging by sharing concrete use cases in the context of bottleneck identification, performance tuning, and bug finding. Finally, we use 3MileBeach to perform a novel type of fault injection - ***T**emporal **F**ault **I**njection* (**TFI**), which more *precisely* controls individual inter-service message flow with temporal prerequisites, and makes it possible to catch an entirely new class of fault tolerance bugs.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Traceability**; • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

Tracing, Temporal Fault Injection, Application Tuning, Bug Finding, Chaos Engineering

ACM Reference Format:
Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3MileBeach: A Tracer with Teeth. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3472883.3486986

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486986

1 INTRODUCTION

Infrastructures that support cross-cutting concerns such as tracing [26, 57] and fault injection [34] have become part of the software quality ecosystem [25] for microservice-based applications. Distributed tracing provides developers and operators with detailed *explanations* of the events during a particular client-level request or system execution. These explanations, which may span both component and layer boundaries, have myriad applications in performance tuning [18, 32, 55] and debugging [28, 52, 65]. While the tracing infrastructure *observes* distributed executions, fault injection frameworks provide support for *perturbing* them by creating or simulating faults such as machine crashes, errors and delays. These frameworks complement existing software quality mechanisms (*e.g.*, continuous integration [59]) that allow developers to rigorously check whether their applications succeed in tolerating these faults.

Unfortunately, supporting tracing and fault injection functionalities can be prohibitively costly in terms of actual expenditures (*e.g.,* for proprietary solutions), runtime overhead, and programmer effort (*e.g.,* making invasive changes to heterogeneous application and infrastructure code). A key challenge is finding suitable interposition points [43, 44, 64] for recording (or perturbing) system state in a landscape in which components are implemented in various languages and communicate via multiple transport mechanisms. To take advantage of tracing and fault injection capabilities, developers have to navigate a trade-off between *precision* (*e.g.,* the quality of the signal during tracing and the granularity at

which faults can be injected) and the *cost* (*e.g.,* programmer efforts and runtime overhead).

We present 3MileBeach, a new distributed tracing and fault injection framework for microservices. 3MileBeach interposes at the message serialization library layer to transparently provide fine-grained tracing and fault injection capabilities. 3MileBeach requires only conditional and limited application-level instrumentation. It provides fine-grained tracing at 25%-50% of the overhead of the state-of-the-art while capturing strictly richer traces that enable new analyses. 3MileBeach *simulates* faults at the granularity of individual requests, supporting massively concurrent failure tests without blast radius [16, 27], making it suitable for testing in production. Finally and most importantly, we contribute TFI, a novel class of fault injection, based on temporal predicates; that can identify bugs that would have lain dormant in the current state-of-the-art.

This paper is structured as follows. In § 2 and § 3, we discuss state-of-the-art tracing and fault injection frameworks, and identify a new class of bug that is, to the best of our knowledge, not addressed by any existing technology. In § 4, we study microservice frameworks (§ 4.1) and discuss how we implemented 3MileBeach with context-propagation [49] mechanisms. In § 5, we measure 3MileBeach's performance as a tracer (§ 5.2), discuss its efficacy as a fault injector (§ 5.3), and show how 3MileBeach provides a platform for rapid innovation of problem localization in different applications (§ 5.4).

2 BACKGROUND

In this section, we review the history of tracing and fault injection infrastructures in the context of modern microservice-based architectures. We show how a key challenge in choosing and deploying both distributed tracing and fault injection is navigating the trade-off between *precision* and *cost* in instrumentation.

2.1 Tracing Frameworks

Despite being only a little more than a decade old, the landscape of tracing infrastructures has grown relatively mature. Nearly all existing systems share a common lineage in XTrace [38], Dapper [62], or both.

XTrace is an early and influential cross-layer tracing framework based on context-propagation to correlate trace entries across layers, machine, administrative boundaries, *etc.* Application and infrastructure code must be instrumented so as to 1) extract context from inputs, 2) possibly update that context with local information, and 3) ultimately include the context with any outputs. A variety of recent systems [45, 48, 50] trace their lineage to XTrace. XTrace and its inheritors represent a trace as an arbitrary ***D**irected **A**cyclic **G**raph* (**DAG**) [33] where labeled nodes represent system events, and edges represent causation.

Dapper was pioneered at Google for the narrower use case of performance tracing for service-oriented architectures. Like XTrace, Dapper also relies on context-propagation. Due to their largely homogeneous infrastructure, Google was able to amortize the effort of instrumentation; *e.g.,* an RPC library that supports thousands of applications need be instrumented only once. Unlike XTrace, Dapper represents a trace as a tree of *spans*, or *annotated intervals* of local computation. This (less general) representation is appropriate to the request-response communication that is common to microservice-based architectures, in which it is reasonable to assume that every request has a response. Most existing open-source tracing frameworks, including X-Ray [1], OpenTracing [14], OpenZipKin [24], and Jaeger [61], are direct descendants of Dapper and inherit its data model and overall architecture.

A significant *cost* of tracing systems is their overhead consisting of the runtime overhead and the instrumentation overhead incurred by programmers to propagate context through applications. Once this instrumentation effort is complete, adding additional instrumentation points is an incremental effort. Still, these high table stakes are a significant deterrent to the adoption of tracing in the first place. Another key weakness of Dapper-based tracing infrastructures is that they are *best-effort* in nature. The reasons are twofold: First, since the collection infrastructures are asynchronous, requested traces may be incomplete because trace components have been delayed or lost; Second, more fundamentally, since span-based models assume two-way communication among nodes in a tree of service calls, they propagate their data only along with request (non-response) flows. Consequently, it is impossible to distinguish a structurally incomplete trace from which some subgraph is missing from a complete trace. Unfortunately, complete traces are needed when it comes to TFI cases, since the *total history* carried by complete traces are the key evidence that help us judge whether the temporal prerequisites are satisfied or not.

2.2 Fault Injection Frameworks

Fault injection has a long history in software engineering, and dependability communities [36, 46, 51], but the use of large-scale fault injection infrastructures for the resilience testing of distributed systems is barely a decade old. As a consequence, the ecosystem is substantially more varied and less mature than that of tracing.

Netflix's Chaos Monkey [42] was a pioneer in ***S**tochastic **F**ault **I**njection* (**SFI**). Chaos Monkey creates a *background radiation* of component failures in production systems by

randomly terminating virtual machines during test executions. Less a testing methodology than a social phenomenon, Chaos Monkey created a culture of resilience. Developers grew to expect remote services to be frequently unavailable, and to mask or mitigate these faults.

Subsequent iterations of SFI developed into a discipline called *chaos engineering* [2, 27, 56]. Chaos Monkey provides a single answer both to the *mechanisms* for fault injection (terminating instances of virtual machines) and to the *strategy* for choosing experiments (*uniformly* and *randomly*), while second-generation infrastructures [23, 53, 54, 60] decouple these concerns, presenting an API to specify individual experiments (*i.e.,* allowing users to specify individual services to *where* inject faults). These systems, in many cases, also improve the *precision* of the fault injectors, targeting individual containers in managed infrastructures; in some cases, individual requests [22, 23] (*i.e.,* specifying *which* interactions to target). SFI approaches also generalize beyond crash faults, simulating delay, network partitions, and explicit transport-level error messages (*i.e.,* specifying *how* to inject or simulate fault events).

In large part, chaos testing approaches have migrated away from the Netflix model of testing in production, combining fault injection with software engineering reliability techniques such as integration tests in staging environments. One example of these ***P**rincipled **F**ault **I**njection* (**PFI**) approaches is Chaos Toolkit [3]. Chaos Toolkit provides a modular automatic framework for controlled chaos experiments. The implementations of fault injectors are factored away into a variety of *drivers*, making it compatible with open source fault injection solutions as well as state-of-the-art proprietary solutions [8]. Unlike the *background radiation* of approaches such as Chaos Monkey, Chaos Toolkit performs one experiment at a time. First, the steady state of the system is observed. In practice, the observation could be as simple as asserting that a particular integration test succeeded, or as nuanced as a collection of key metrics (*e.g.*, system throughput, latency distributions, error counts, *etc.*) with tolerances. Then the system is *perturbed*, typically by effectuating systems faults via a *driver*, and the observation is repeated to see if there is a delta. Finally, service is fully restored by terminating the fault injection. The PFI approach adopted by systems like Chaos Toolkit makes it possible to explore the space of possible fault experiments systematically, and in some cases, to make the effects of experiments *reproducible*.

Instead of crashing instances or virtual machines, ***R**equest **L**evel **F**ault **I**njection* (**RLFI**) approaches like ALFI [22] and FIT [23] *simulate* faults during fault injection tests. RLFI approaches subsume and outperform Chaos Toolkit since RLFI approaches also determine the faults to be simulated on a per-request basis while supporting *concurrent* fault injections.

In Table 1, we summarize and categorize the criteria of the aforementioned fault injection frameworks into five main aspects.

Table 1: Taxonomy of Fault Injection Frameworks. Five criteria: C_1) the ability to support concurrent experiments; C_2) avoid blast radius; C_3) trigger TFI bugs; C_4) reproduce bugs; and C_5) avoid application-level instrumentation.

Approach	C_1	C_2	C_3	C_4	C_5
SFI	×	×	✓[1]	×	✓
PFI	×	✓	×	✓	✓
RLFI	✓	✓	×	✓	×
TFI	✓	✓	✓	✓	✓[2]

The varied landscapes of tracing and fault injection suffer from many similar problems. The interposition required to realize fault injection often has prohibitive *costs*. Unlike tracing, after a one-time instrumentation effort adding additional finer-grained details has incremental *costs*, high *precision* fault injection (specifying *which* interactions to interpose on) is only available as a proprietary technology. Unfortunately, as described in § 3, even the most sophisticated fault injectors still leave an important class of bugs on the table.

3 MOTIVATION: A TIMING PROBLEM

The tracing and fault injection ecosystems are very different despite of their similar mechanisms. Costs aside, trace collection infrastructures are relatively mature and provide homogeneous capabilities while fault inject infrastructures remain in flux.

The current state-of-the-art systematic fault injection for distributed systems has suffered from two fundamental problems. First, while coarse-grained mechanisms that crash individual virtual machine instances or containers are widespread, users who wish to enjoy the benefits of simulation, concurrent experimentation, and freedom from a blast radius must turn to proprietary solutions. Second, there is an essential class of bugs that no existing fault injection technology is expressive enough to discover to the best of our knowledge.

[1] The randomness of Chaos Monkey in principle triggers TFI-like faults, but lacking *logical* timing (discussed in § 3.2) makes the results not so reproducible.

[2] 3MileBeach still needs one-time efforts to modify 1) application code that import packages, and files that manage dependencies (*e.g.,* pom.xml, build.gradle, requirements.txt, *etc.*), making 3MileBeach the dependency; 2) generated files of serialization libraries (*e.g.,* Protocol Buffers [15], Flutter [12], *etc.*), making messages carry 3MileBeach's payloads; 3) application code if and only if a little portion of libraries still require semantic but tracing-unrelated modifications.

3.1 TOCTTOU Bugs

Consider a service, Frontend, that orchestrates an electronic commerce application. Frontend is invoked when a user is ready to purchase the items in their cart. First, Frontend sends a message to CardHandler, a downstream service that validates the user's credit card information (locally, in the event of a cache hit, or by invoking a third-party billing service). If this succeeds, Frontend sends a series of messages to its downstream services like Product to calculate the total purchasing price, and CurrencyService to convert currency if necessary. Otherwise, Frontend tries to validate the next card, or prompts the user if it cannot find one. Finally, Frontend sends a message to CardHandler again to bill the user's credit card.

Imagine a latent bug exists in Frontend: an incorrect expectation that the second request to CardHandler (for billing) would succeed if the first request to CardHandler (for validation) succeeded. The programmer anticipated that the third-party billing service might be unavailable, but has committed a ***T**ime **O**f **C**heck **T**o **T**ime **O**f **U**se* (**TOCTTOU**) error [17, 29, 30]. This assumption is easily violated since 1) the third-party service became unavailable after the first request; 2) more likely, the third-party service was unavailable all along, but a cache hit allowed the validation step to succeed. During violation, the application returns unintelligible errors to users that are unsure of whether their card has been billed or not. Users will retry and may lead to double payment.

This example is oversimplified. However, similar hazards exist in any application where a given microservice is invoked more than once in the life cycle of a particular client-level request. Additionally, this example is drawn from a large space of possible bugs outside the purview of existing fault injection techniques. To understand this space, we need to know how individual services in a microservice-based application can be sensitive to faults in other services that they depend on, and to what extent this sensitivity can be time-varying.

3.2 Temporal Discretization

Systematic fault injection tools such as Chaos Toolkit do not consider faults in time dimension. Instead, they assume that faults either happen (or don't) during the life cycle of a client-level request flow, as opposed to at an arbitrary time in the request flow. This assumption prevents such approaches from reaching TOCTTOU bugs. Unfortunately, adding a dimension of time to fine-grained fault injectors by making them specify *when* in addition to *which*, *how*, and *where* is doomed from the start. First, time is continuous, and the fault space is already too large to exhaustively explore! Second, time is an elusive notion in distributed systems; it would not be meaningful (or repeatable) for a fault injector to be directed to inject a fault at a particular service at an exact *real* time.

Hope is not lost. Triggering timing-related bugs in a given application never requires sweeping the (infinite) space of *real* time. Our solution to this problem is to divide *real* time into a (small) set of discrete equivalence classes such that injecting any two faults in the space (*e.g.,* the same faults at different *real* times) would be indistinguishable to the service(s) under test.

Consider the example described in § 3.1. The TOCTTOU bug in Frontend can be triggered by the injection of a single service crash. Unfortunately, the temporal discretization [19, 35] taken by existing fault injectors, where faults are injected strictly before the test or randomly during the test, is too coarse-grained to trigger the bug. Note that the sentinel event that separates the interval during which the crash of CardHandler will be tolerated by Frontend is the completion of Frontend's first call to it. That is, crashes will be tolerated before this event but not after. What's more, this example generalizes communication events - sending and receiving of messages to downstream services, punctuating the otherwise continuous time dimension.

Based on this insight, even if treat Frontend as a *black box*, we can still observe four discrete *logical* time (t) intervals externally where CardHandler may crash. As shown in Figure 1, these intervals are:

1) before Frontend's first request ($t < t_0$);
2) between Frontend's first request and CardHandler's first response ($t_0 \leq t < t_1$);
3) between CardHandler's first response and Frontend's second request ($t_1 \leq t < t_2$);
4) between Frontend's second request and CardHandler's second response ($t_2 \leq t < t_3$).

Any two crashes in the same *logical* interval *look the same* to Frontend, and hence will be sufficient to trigger *any* temporal-related bugs, not just this one!

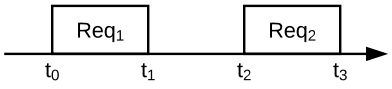


Figure 1: Timeline. A service processes two requests (Req_1 and Req_2) in a client-level request flow.

3.3 Communication is The Thing

The critical insights behind TFI are the following. First, an upstream service may tolerate faults in a downstream service for parts of a request lifetime and fail to tolerate them at another time. However, as shown by Figure 1, the space of *logical* times at which a service's tolerance of downstream

bugs might change is far from infinity in practice. Injecting a given fault at every *logical* state of service's implicit state machine [47, 58] is sufficient to exercise every possible behavior in response. Each received message triggers a state transition, and each sent message is caused by a state transition. We can approximate the opaque state machine of microservices from the outside by simply interposing on a system at a place that has a purview into communication. From 3MileBeach's perspective, two states are considered the same when they obtain the same communication (received-sent message) patterns.

3MileBeach's tracing functionality correlates effects with causes *across* machine boundaries and in the face of *concurrency*, while its fault injection functionality *simulates* faults that the software is expected to tolerate. When choosing interposition points in an arbitrary distributed system, the fundamental difficulty is the trade-off between minimizing the *cost* and maximizing the *precision*.

In a microservice-based architecture, however, there is a convenient *choke point* located precisely at the boundary between communication components: the libraries that are used to serialize (and deserialize) message payloads to (and from) wire formats such as JSON [11] and Protocol Buffers. In practice, 3MileBeach takes communication as interposition point that requires a one-time effort of modifying serialization libraries (one for each language-format) and limited (*i.e.*, tracing-unrelated application- or infrastructure-level) instrumentation. This choice gives us an economy of mechanism for tracing and fault injection. 3MileBeach adorns messages with *trace metadata*, as the state-of-the-art tracing systems do, propagates each outgoing message the context of the inbound request or requests that directly lead to it. 3MileBeach also takes advantage of this mechanism to propagate *fault injection configurations* that allow us to target particular requests during fault injection. After all, faults in remote services are invariably witnessed at component boundaries, and are manifested as delay, explicit errors, or corrupt messages [31]. There is no blast radius of faults since the simulations are confined to a particular client-level request context, which permits lightweight and concurrent tests.

Finally, interposing based on context-propagation mechanisms allows us to devise the first practical fault injector to realize TFI. Our modifications to the third-party libraries permit context-propagation on request and response messages. Therefore, whenever a message of any kind is received, it carries the complete causal history of service calls carried by *trace events* witnessed by its sender, associated with the current client-level request. Fault injection logic then inspects the causal history to determine:

1) whether the given request has been singled out for injection (*which*);
2) whether the current service should simulate the fault (*where*);
3) what kind of fault to simulate (*how*);
4) and finally, *when* we simulate the fault (now or later?).

4 IMPLEMENTATION

During implementation, we studied different design patterns of microservice frameworks. We chose a common context-propagation mechanism (§ 4.1) and applied data structures to carry *3MileBeach's payload* (***3mb-payload***) to support tracing and fault injection functionalities (§ 4.2). In § 4.3, we introduce the algorithms we apply to collect traces and to inject faults.

4.1 Architecture Abstraction

The microservice architecture [37] is designed to separate application features into independently implemented services that communicate using well-defined APIs. A natural approach to manage complexity is using a microservice framework that abstracts message transport mechanisms, microservice placement on both physical and virtualized resources, and the infrastructure itself [4, 13]. However, the abstractions that microservice frameworks provide also contribute to significant heterogeneity amongst applications.

3MileBeach takes advantage of the constraints of the microservice architecture to provide tracing and fault injection with only one assumption about microservice implementations or infrastructure. That is, a microservice framework provides each microservice with a plumbing abstraction such as service handlers that allow services to associate logic (*e.g.*, handler function) with inbound and outbound messages (*i.e.*, requests and responses). In Figure 2, we refer to the components that lie adjacent to the boundaries between the service handler and the microservice frameworks as *boundary components* (*i.e.*, inbound function and outbound function).

Despite the distinct framework implementations and application programming languages, every microservice framework provides mechanisms to link service handlers to the boundary components. Panorama [40] introduced four design patterns of *component interactions*[3] by abstracting observability as *direction* which directly calls handler functions, and *indirection* that asynchronously invokes handler functions with (*in-*/*out-*) queues/proxies as cache layers. During our study of microservice frameworks, we generalized the four design patterns:

P_1) **Full direction.** All functions are assigned to a single thread where the inbound component, service handler, and the outbound component are sequentially called;

[3] In this paper, we use *communication*.

P_2) **Inbound indirection.** *In*-queue stores inbound messages before the worker thread picks one, calls the service handler, and then invokes the outbound component;

P_3) **Outbound indirection.** Inbound component and the service handler are called directly in a worker thread while the outbound messages are sent to *out*-queue for further network transportation;

P_4) **Full indirection.** The worker thread picks inbound messages from *in*-queue, invokes the service handler, and sends the outbound message to *out*-queue.

Patterns P_2, P_3 and P_4 apply *explicit* context-propagation mechanisms that pass essential identification information through boundary components and the service handler, *regardless of the implementation of the service handler*. The remaining pattern, **Full Direction**, can easily link inbound and outbound messages with the service handler via *thread metadata* [63] since boundary components and the service handler are executed in the same thread. As a simplification, we claim that P_1 also performs a *shadow* and *implicit* context-propagation mechanism. Thus, we can apply 3MileBeach to aforementioned design patterns through existing context-propagation mechanisms.

Figure 2 provides a high-level view of service handlers, boundary components, and inbound/outbound messages. 3MileBeach takes advantage of third-party libraries' existing context-propagation mechanisms to abstract the patterns mentioned above as a cell model, upon which tracing and fault injection functionalities are implemented. In general, the inbound component deserializes raw inbound messages extracted from the network, and invokes the service handler. The outbound component processes outbound messages from the service handler, then sends the data as outgoing messages. The participation of boundary components on every request and response makes them ideal interposition points without *application*-level knowledge. 3MileBeach collects and propagates *3mb-payloads* by interposing the boundary components and treating service handlers as *black boxes*.

Base on these abstractions, we demonstrate two typical data flows, ***D**irect **R**esponse **C**ircle* (**DRC**) and ***S**ynchronized **R**equest-Response **C**ircle* (**SRC**), in an abstracted architecture of microservice frameworks and service handlers. Libraries (*e.g.,* gRPC [9] and language-specific ones such as Go Gorilla [7]), deserialize inbound messages from the network before invoking a service handler and serialize the outbound message into wire format. Both deserialization and serialization are done by serialization libraries (*e.g.,* Protocol Buffers, JSON, *etc.*).

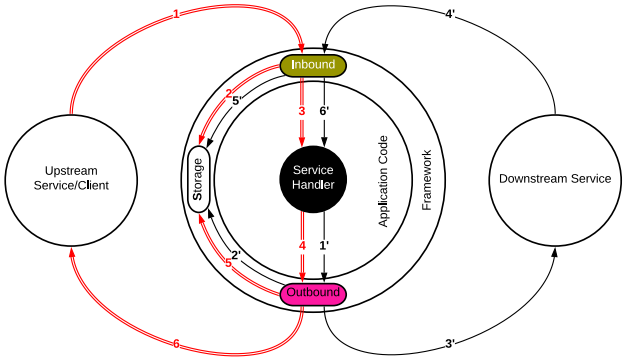


Figure 2: Cell Model. Direct Response Circle (⇒) and Synchronized Request-Response Circle (→).

4.2 Data Structure

3mb-payload is concretely represented by **Trace**, which is the top-layer data structure when we interpose on the third-party libraries. **Trace** consists of a list of *event-s* (**Event-s**), essential *trace metadata* (*e.g.,* **ID**) that helps 3MileBeach identify and assemble events of a specified trace, and a list of ***f**ault **i**njection **c**onfiguration-**s*** (**FIC-s**).

4.2.1 Event. An instance of **Event** records essential information of an event, that is, when (**Timestamp**) a **Service** receives or sends (**Action**) a request or response (**Type**) with a certain name (**Name**). 3MileBeach has various methods to fetch **Type** and **Name** (*e.g.,* from type definitions, message description files, *etc.*), depending on the libraries and programming languages. **UUID**, which is generated automatically by the requester's outbound component, helps 3MileBeach identify and assemble all the events related to a particular SRC. Requester's and responder's boundary components sign **Service** with the services' signatures (in the absence of programmer annotations), which are usually UUIDs.

As shown in Figure 3, there are four events associated with the same **UUID**, indicating the life cycle of a single **SRC**. Two of them are recorded by the requester (Svc_1) when it sends a request (Req_1) at t_1 and receives the corresponding response (Rsp_1) at t_4, and the other two events are recorded by the responder (Svc_2) at t_2 and t_3.

4.2.2 Fault Injection Configuration. We use ***f**ault **i**njection **c**onfiguration* (**FIC**) to describe the objective of a TFI or RLFI test case. In the remainder of the paper, we use n to represent the number of services within a certain microservice-based application. Consider an application consists of n services, namely $Svc_0, Svc_1, \ldots, Svc_{n-1}$.

RLFI injects faults to services *prior to* the life cycle of a client-level request. We use **FIC**{**Type**: Crash, **Name**: Svc_i}

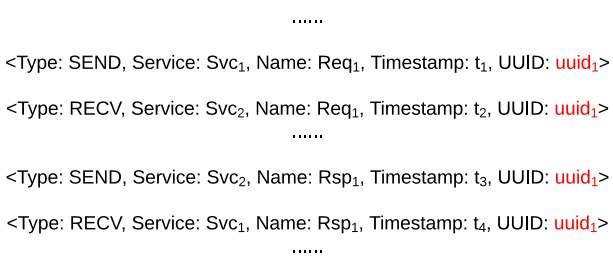


Figure 3: Four Events Associated with an SRC.

to denote simulating crashes involving Svc_i. To test all patterns of crashing services, the space of RLFI experiments consists of a power set of 2^n cases in total[4]. However, when we are testing a particular client-level request, we will not invoke all the services of the *application*. Say, if m out of n services will not be invoked, we might waste time performing experiments that involve irrelevant services. We can reduce the total number of test cases from 2^n to 2^{n-m} by preventing the fault injection framework from affecting unrelated services, which reveals an immediate benefit of closely coupling tracing and fault injection infrastructures. In many cases, a single integration test will suffice to enrich the number of relevant services to project future fault injection experiments.

Unlike RLFI that determines the status of services at the beginning of a test, TFI simulates faults during a certain test execution according to *logical* timings. During a TFI test case, 3MileBeach determines the behavior of the responder(s) when the requester sends each *individual* request. We use **After**, a list of **TFIMeta**s, to store all the temporal prerequisites of the faults. The space of faults that TFI can trigger is a superset of that of RLFI faults since RLFI is a special case of TFI when **After** is empty. More importantly, **FIC** massively reduces the space of TFI tests by applying temporal discretization. As shown in Figure 1, suppose Svc_1 receives Req_1 and Req_2 from Svc_0 in processing a client-level request. RLFI only determines whether the two requests both succeed or fail. If we want to fail Req_2 without affecting Req_1, we should simulate the fault at any *real* time interval (t_{start}, t_{end}) where $t_1 \leq t_{start} < t_2$ and $t_3 < t_{end}$. Since time is continuous, we will get a infinite (t_{start}, t_{end}) set. TFI shrinks the space of the (t_{start}, t_{end}) set by simulating the crash exactly at time t_{send} $(t_1 < t_{send} < t_2)$ when Svc_0 sends Req_2 to Svc_1. To describe this case, we can use **FIC**{**Type**: Crash, **Name**: Req_2, **After**: [**TFIMeta**{**Name**: Req_1, **Times**: 1}]}. 3MileBeach does not assign the *entire* trace to the *request* messages to guide fault injection. Instead, an evidence of whether Req_1 *has been called* will be attached to the *request* messages since downstream services only need the information that is *relevant to* the **FICs**.

It is not practical to create **FICs** for complex applications by hand. During evaluation, we used a thick client to exhaustively generate **FICs** within RLFI or TFI space. In the future, we will offload the problem of experiment selection to an offline tool such as LDFI [20, 21] to get more efficient and precise experiment guidance.

4.3 Algorithms

In this section, we discuss how 3MileBeach interposes on the boundary components by making serialization functions operate *3mb-payloads* (§ 4.3.1), and the role of the re-written serialization functions in the data flows (§ 4.3.2).

4.3.1 Interpose via Serialization Functions. All messages sent and received among services are transformed by serialization functions (referred as Deserialize and Serialize) to and from wire formats. To trace the services that participate in processing a client-level request, 3MileBeach extends the serialization functions. The names of the extended serialization functions are Deserialize' and Serialize'.

Through a service's local storage S, 3MileBeach is able to link inbound and outbound messages regardless of the service handler. 3MileBeach uses S to store a *trace* associated with a context object, Ctx. Ctx is derived from the *existing* context-propagation mechanism that carries *request metadata*. 3MileBeach makes Ctx carry *trace metadata* represented by **ID**, which is initialized by the client. In the inbound component, 3MileBeach gets **ID** from the inbound messages and assigns **ID** to Ctx (#L13, Algorithm 1). When the service handler sends messages, 3MileBeach takes advantage of context-propagation mechanisms to add the obtained **Trace** (stored in S) to the outbound messages (#L8-9, Algorithm 2). In Table 2, we describe some essential functions.

4.3.2 Serialization Functions and Data Flows. In the following, we rephrase Algorithm 1 and 2 in terms of the data flows discussed in Figure 2, and describe the four distinct pairings of *action* (SEND and RECV) with message *type* (REQUEST and RESPONSE).

Direct Response Circle (DRC).

~~1~~ Framework RECVs a REQUEST from an upstream service or the client;

~~2~~ Inbound component invokes Deserialize' which calls Deserialize to deserialize the REQUEST (#L3), extends the *trace* by recording the *event* (#L10), stores the *trace* to S (#L11), and assigns *trace metadata* to Ctx (#L13);

~~3~~ Framework invokes service handler and waits until the life cycle of the invocation ends;

[4]For simplicity, we always count the case that injects no faults.

Table 2: Descriptions of Essential Functions

Name	Description
GetUUID	Gets *uuid* according to *act*(*ion*) − *type*: 1) **SEND-REQUEST**. On capturing the first event of an SRC, GetUUID generates a new *uuid* for the requester; 2) **The Rest**. The *uuid* can be fetched from *trace*.**Events** recorded by the other service that participates in the same SRC.
trace.Extend	Extends the input *event* to *trace*.**Events** and updates *trace*.**FICs**.
S.ExtendTrace	Gets *trace* by **ID**, calls *trace*.Extend(*event*), and eventually returns *trace*.
S.SetTrace	Upserts *trace* according to *trace*.**ID**. More specifically, during updates, *S* merges the **Events** from both *traces* and recalculates *trace*.**FICs**.

Algorithm 1 Deserialize'

```
1:  act = RECV
2:  function Deserialize'(data)
3:      msg = Deserialize(data)
4:      if msg contains a valid trace then
5:          trace = msg.Trace()
6:          t = CurrentTimestamp()
7:          type, name = msg.Type(), msg.Name()
8:          uuid = GetUUID(act, type, trace)
9:          event = Event(t, act, type, name, uuid)
10:         trace.Extend(event)
11:         S.SetTrace(trace)
12:         if type == REQUEST then
13:             Ctx.ID = trace.ID
14:     return msg
```

Algorithm 2 Serialize'

```
1:  act = SEND
2:  function Serialize'(msg)
3:      if Ctx.ID associated with a trace from S then
4:          t = CurrentTimestamp()
5:          type, name = msg.Type(), msg.Name()
6:          uuid = GetUUID(act, type, trace)
7:          event = Event(t, act, type, name, uuid)
8:          trace = S.ExtendTrace(Ctx.ID, event)
9:          msg.SetTrace(trace)
10:         if type == RESPONSE then
11:             S.DeleteTrace(Ctx.ID)
12:         else if type == REQUEST then
13:             err = SimulateFault(trace)
14:             if need to return err then
15:                 return err
16:     data = Serialize(msg)
17:     return data
```

~~4~~ Framework gets the RESPONSE;
~~5~~ Outbound component invokes Serialize' that retrieves *trace* from *S* with the *trace metadata* from *Ctx* and extends the *trace* by recording the SEND *event* (#L8), deletes *trace* (#L11), and calls Serialize (#L16) to serialize the RESPONSE;
~~6~~ Framework SENDs the RESPONSE back to the upstream service or the client.

Synchronized Request-Response Circle (SRC).

~~1~~ Service handler SENDs a REQUEST to the downstream service through a *blocked function call*;
~~2~~ Outbound component invokes Serialize' that retrieves *trace* from *S* with *Ctx* which has already been stored when the microservice RECVed a REQUEST from the upstream service or the client, extends the *trace* by recording the SEND *event* (#L8), *simulates* fault (#L13), and calls Serialize to serialize the REQUEST if no error codes are returned to the service handler (#L16). The following steps are taken when #L15 has *not* been reached;
~~3~~ Framework SENDs the REQUEST to the downstream service and *waits for the RESPONSE*;
~~4~~ Framework RECVs the RESPONSE;
~~5~~ Inbound component invokes Deserialize' that calls Deserialize to deserialize the RESPONSE (#L3), extends the *trace* by recording the RECV *event* (#L10), and stores the *trace* to *S* (#L11);
~~6~~ Service handler will RECV the RESPONSE as a return value of a *blocked function call*.

4.3.3 Fault Simulation. 3MileBeach performs fault injection by simulating the externally visible effects of a downstream fault that caused by network transportation or by the responder handler, from the perspective of the requester. Since 3MileBeach does not actually crash and restore services during fault injection, we can test multiple fault injection cases concurrently and control blast radius.

A typical SRC contains two services (*i.e.*, a requester and a responder) and two data flows (*i.e.*, a request flow and a response flow). When a fault is triggered, the requester is unaware of the root cause of the downstream fault; instead, depending on its implementation, it will witness the fault through status codes or return values such as timeout, connection closed, package loss, *etc.* What's more, we can better

control the behavior of service handlers than that of data flows since 3MileBeach works upon the boundary components lie between services and data flows.

Function SimuateFault mentioned in Algorithm 2 reads every **FIC** from **FICs**. 3MileBeach will trigger faults when the prerequisites defined by **FICs** are satisfied. Since 3MileBeach *simulates* faults when the requester sends requests, user written error handlers will receive the exact error codes like timeout or service crash, and cannot distinguish whether the error codes are caused by the simulated faults or by the actual ones. Once requesters have collected new *events* (#L11, Algorithm 1 and #L8, Algorithm 2), 3MileBeach recalculates **After** by refreshing every **TFIMeta** according to newly captured **Events**.

5 EXPERIMENTAL STUDY

In this section, we describe our experimental environment (§ 5.1), measure the ***E**nd-**T**o-**E**nd* (**E2E**) latency overhead of 3MileBeach as a tracer in comparison to the state-of-the-art (§ 5.2), demonstrate the efficacy of 3MileBeach as a fault injector (§ 5.3), and provide two examples of localizing problems (§ 5.4).

5.1 Environment Setup

We applied 3MileBeach to Hipster Shop [10], a microservice demo application (the *application*) that we deployed on Google Kubernetes Engine (GKE) [6]. A *client* program (the *client*) generates test cases for the *application* to help us perform tracing and fault injection, apply performance tuning, and locate bugs.

5.1.1 Application. Hipster Shop consists of ten services: Frontend (Svc_{FE}), Cart (Svc_{Cart}), Recommendation, ProductCatalog (Svc_P), Shipping, Currency (Svc_C), Payment, Email, Checkout, and Ad (Svc_A). Hipster Shop has a moderate complexity of services and topology, making it an viable platform to test 3MileBeach's functionality and performance. The data serialization, transportation, and framework libraries (JSON, Protocol Buffers, RESTful, GRPC, Gorilla, *etc.*) applied in Hipster Shop are universal among microservice-based applications. Hipster Shop's services are written in five languages (Go, C#, Node.js, Python, and Java), which helps testing the language generality of 3MileBeach.

5.1.2 Clusters. As shown in Table 3, nodes of clusters are chosen from GKE's first generation general-purpose machine types (n1-standard family). **Cluster1** and **Cluster2** have 16 vCPUs and 60GB of memory individually. Both clusters are deployed in the same zone (us-west1-b).

5.1.3 Client. The *client* sends requests to Svc_{FE} under various levels of concurrency. In this paper, we use N to denote concurrency (*i.e.,* $N = 1, 2, 4, ..., 128$). We deploy the *client* in the same cluster as the *application* to minimize network latency between the *client* and Svc_{FE}.

Table 3: Cluster Configurations

Name	Machine Type	CPU	Mem	#
Cluster1	n1-standard-2	2	7.5GB	8
Cluster2	n1-standard-16	16	60GB	1

During test executes, the *client* first generates *3mb-payloads* that contains *trace metadata*, empty **Events**, and particular **FICs**. Then *3mb-payloads* are injected to the request message sent by the *client*. Finally, the *client* collects traces and generates reports.

5.2 Tracing Benchmark

It is unavoidable that we impose some overhead when collecting traces, and we must be careful not to impose too much of a penalty when inspecting an application. To study the imposed overhead of 3MileBeach as a tracer, we collected tracing benchmarks of 3MileBeach and compared against Jaeger, a popular tracing framework, and the instrumented application.

5.2.1 Test Cases. The *client* generates requests with *3mb-payloads* injected and sends them to */home*, an endpoint of Svc_{FE}. To satisfy a client request, Svc_{FE} generates calls to four downstream services that will respond to five GRPC requests including Svc_A's GetAds (Req_{A1}), Svc_{Cart}'s GetCart, Svc_C's GetCurrencies (Req_{C1}) and CurrencyConversion (Req_{C2}), and Svc_P's ListProduct (Req_{P1}).

As a Dapper-style tracer, Jaeger requires initialization and annotations in the application-level code to define spans, and collects or stores traces locally before pushing them to a centralized database. Since Jaeger does not collect spans from Svc_{Cart}, to get an apples-to-apples comparison, 3MileBeach will not collect events from Svc_{Cart}, either. However, 3MileBeach is still able to simulate faults to Svc_{Cart} due to the interposition point which is located at Svc_{FE}'s outbound component.

We ran the *application* under three different settings: with 3MileBeach enabled (3MileBeachOn) as the tracer, with Jaeger enabled (JaegerOn), and a baseline with no tracer enabled (TracerOff). Each setting was executed in multiple rounds to calculate an averaged result with a corresponding standard deviation. The *client* sent out a total of 131,072 ($= 1,024 * N_{max} = 1,024 * 128$) requests during each round. This allowed us to directly compare the overhead of 3MileBeach to that of Jaeger in terms of baseline. To get a fair comparison, we made Jaeger's spans only carry the same amount of information (*i.e.*, message name, action, type, timestamp, *etc.*) as 3MileBeach's **Event** does.

5.2.2 Throughput and End-to-End Latency. Figure 4 shows how throughput and E2E latency changes in response to more workload. 3MileBeach outperformed Jaeger in all cases. When N was 128 on Cluster2, both tracers reached the worst case where the overhead of Jaeger climbed rapidly to 168.4% while that of 3MileBeach grew smoothly to 42.1%, and Jaeger only reached one-third of the throughput in comparison to the baseline. Note that there exists a trade-off between higher message overhead and fewer network transportation. The former causes throughput loss while the latter helps improve the performance. 3MileBeach makes the data flow carry traces, which enables the *client* retrieve traces from the responses. Jaeger generates and tries to submit spans when witnessing the events. Although Jaeger has pool or buffer that reduces the number of network transportation, it still requires additional *postmen* to carry traces, which leads to more messages. This result showed that the number of messages sent contributes more to throughput overhead than larger message size does, which allows 3MileBeach to maintain high performance.

5.2.3 Overhead and Sample Rate. 3MileBeach supports trace sampling. The sample rate determines the percentage of client-level requests that will be traced. To understand the impact of higher or lower sample rates on overhead, we fixed N to 128 and varied the sample rate from 1/1 to 1/64. As shown in Table 4, we can cut the overhead in half by lowering the sample rate from 1/1 to 1/4. Furthermore, if we lower the sample rate to 1/256, we can barely observe overheads.

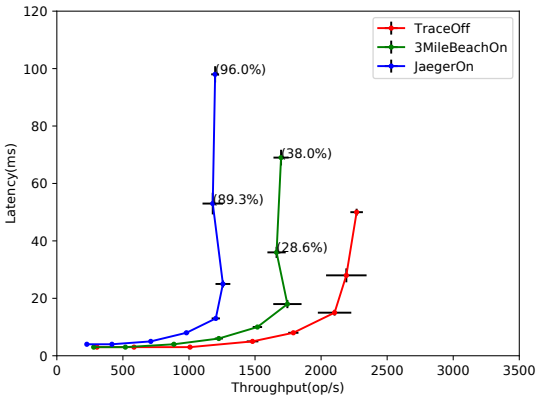
Table 4: End-to-End Latency Overhead. The baselines of Cluster1 and Cluster2 are 56.3 ms and 38.0 ms.

(a) Cluster1		(b) Cluster2
Sample Rate	Overhead	Overhead
1/1	19.9%	46.3%
1/4	7.45%	18.4%
1/16	6.97%	14.1%
1/64	5.33%	7.34%

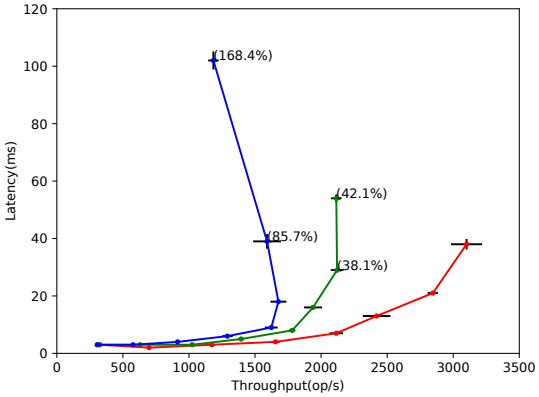
5.3 Fault Injection

To demonstrate how quickly 3MileBeach can sweep the space of TFI test cases, we performed a set of experiments designed to reach bugs in the *application*. We deliberately left two bugs in Svc_{FE}:

DeepRLFI. DeepRLFI can be reached without temporal conditions, and can be triggered if and only if both Svc_A and Svc_C are down, that is, we need to apply *3mb-payloads* that crash both Svc_A and Svc_C (Figure 5c). Any other *3mb-payloads* that only involving Svc_A (Figure 5b) or Svc_C (Figure 5a) individually cannot reach DeepRLFI.



(a) Cluster1



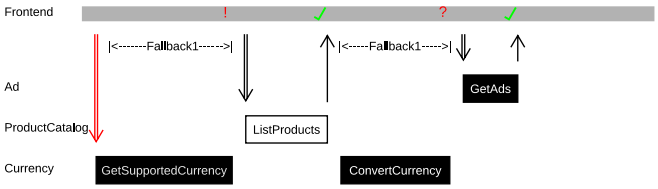
(b) Cluster2

Figure 4: Throughput-Latency of Three Settings. Overheads are shown in parentheses.

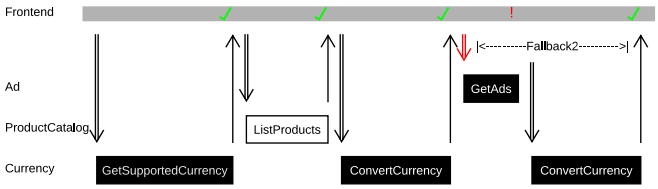
SimpleTFI. SimpleTFI is a TOCTTOU bug similar to what has been described in § 3.1. To trigger SimpleTFI, we need to let requests carry *3mb-payloads* that fail Svc_C after it has received *the second* request during processing a client request (Figure 5d).

5.3.1 Test Cases. The *client* generates *3mb-payloads*, and detects 500-type errors for three fault injection strategies:

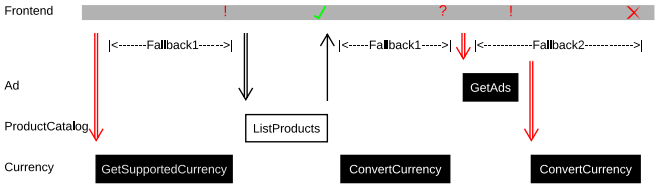
Targeted-Exhaustive RLFI. The *client* exhaustively simulates faults (*i.e.*, crashes) among four microservices that are related to endpoint */home*, which covers the combinatorial space of 16 tests. This targeted approach of pruning the space of experiments is enabled by the mechanism of 3MileBeach that combines tracing and fault injection. We can use the same infrastructure to "turn on the lights" and get an idea



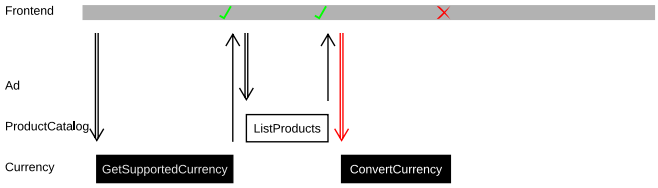
(a) During the first call to Svc_C, Svc_{FE} is fault-tolerant: if Svc_C is accessible, Svc_{FE} assumes that Svc_C will be accessible at all time; if not, Svc_{FE} will apply a fallback strategy (Fallback1) that uses default currency and prices. If we apply RLFI to simulate crashes involving Svc_C, the *client* can still get proper results.



(b) When Svc_A is not accessible, Svc_{FE} will apply a fallback strategy (Fallback2) that uses the default advertisement and sends Svc_C a request to convert the currency. If we apply RLFI to simulate crashing Svc_A, the *client* can still get proper results.



(c) During Fallback2, Svc_{FE} will not check the accessibility of Svc_C anyway, no matter whether Fallback1 has been applied or not. When we apply *3mb-payloads* that simulate crashing services including Svc_A and Svc_C, the *client* will receive a 500 error.



(d) When we apply *3mb-payloads* that simulate crashing Svc_C during Svc_C's second invocation, the *client* will receive a 500 error due to the TOCTTOU fault in Svc_{FE}.

Figure 5: Fault Injection Tests. (✓: request succeeds; !: the requester handles responder's failure via fallback strategies; ?: fallback strategies do not send requests to the responder; ×: the requester can not handle responder's failure.)

of what services may *actually* be involved in a particular client request by tracing the system before performing fault injection experiments. In practice, the four relevant services might be selected by an expert, either human or AI, who understands the semantics of the system.

Exhaustive RLFI. Different from the behavior of executing Targeted-Exhaustive RLFI tests, the *client* exhaustively and "blindly" simulates faults among all possible nine microservices in the *application* with Svc_{FE} excluded. Exhaustive RLFI covers the combinatorial space of 512 tests, containing the aforementioned 16 Targeted-Exhaustive RLFI tests.

Targeted-Exhaustive TFI. During a normal process of the endpoint */home*, there will be 14 requests sent among four services. In this case, the *client* exhaustively simulates faults to the 14 requests with temporal conditions, covering the combinatorial space of 16384 tests, including 16 Targeted-Exhaustive RLFI tests.

5.3.2 Performance. During performance test, we applied 3MileBeach and ran Exhaustive RLFI, Targeted-Exhaustive RLFI and Targeted-Exhaustive TFI on **Cluster2**, and recorded *3mb-payloads* that caused 500-type errors. As a comparison, we applied another round of Targeted-Exhaustive and Exhaustive RLFI by using command-line tools (Command) to crash and restore microservices. Command, including *docker* [4] and *kubectl* [13], are more easily controlled while providing similar functionalities to Chaos Monkey and Chaos Toolkit. Algorithm 3 shows how Command performs a fault injection execution through given **FICs**. Each run started from a *steady state* (#L2: all services are ready) to another steady state (#L11). We used command "*docker service scale <SERVICE-ID>=0*" to shut down a service (#L4), and command "*docker service scale <SERVICE-ID>=1*" to restart a service (#L7). A request was sent by the *client* after all the targeted services had been shut down, and before recovery. During execution, we executed command *kubectl get pods –all-namespaces* to check the status of all services. Redeployment would be triggered when services had failed to restart, or other services had been affected by blast radius.

Table 5 shows how 3MileBeach's advantages in crash simulation and concurrency helped 3MileBeach dramatically outperform Command. Command executed test cases sequentially. It usually took Command 25 to 30 seconds to finish a test case that didn't run into unexpected failures (*e.g.*, services failed to restart, or affected by blast radius, *etc.*). Furthermore, from the "Targeted" test cases, we can see the improvement when *3mb-payloads* are derived from expertise.

5.4 Localizing Issues: Two Stories

To demonstrate how the techniques described in § 5.2 and § 5.3 enable rich application-specific analyses, we introduce

Algorithm 3 Command

```
1: function Command(fics)
2:     t_start = now()
3:     for each fic in fics do
4:         Shut down service fic.Name
5:     Client sends a request, and records status code
6:     for each fic in fics do
7:         Restart service fic.Name
8:     if Services fail to restart, or errors happen in other services then
9:         Redeploy the application
10:    Wait until all services are ready
11:    t_end = now()
```

Table 5: Performance of Fault Injection. T and E refer to Targeted and Exhaustive, respectively.

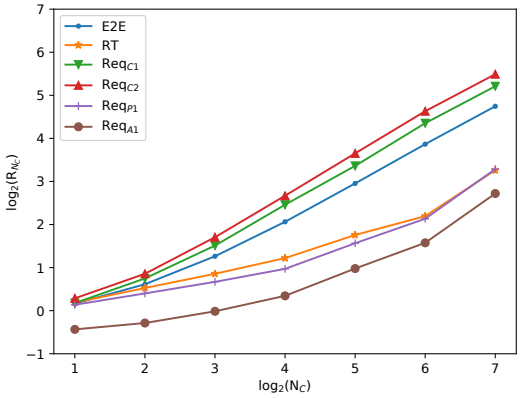
Type	Cases	3MileBeach	Command
T-E RLFI	16	200ms	9m34s
E RLFI	512	5s	5h21m
T-E TFI	16384	90s	N/A

two stories to demonstrate how to localize performance and functionality problems.

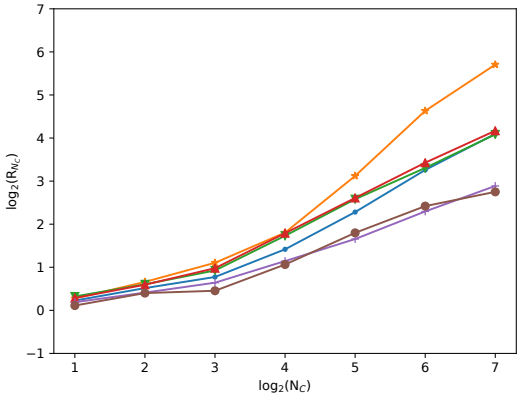
5.4.1 Performance Tuning. *3mb-payloads* collected by the *client* allow us to more precisely and carefully study the latencies of requests in the *application*. This section describes a story of identifying potential optimization targets by rerunning 3MileBeachOn and allowing the *client* to deserialize and study the traces. We applied this to Hipster Shop and were able to identify bottlenecks that we optimized to improve performance.

The latencies we collected were the E2E latency between when the *client* sends a request and receives response, the ***R**ound-**T**rip* (**RT**) latency between the *client* and Svc_{FE}, and the processing latencies measured by request handlers Req_{A1}, Req_{C1}, Req_{C2}, and Req_{P1}. We denote the average measured latency of type T (*e.g.*, RT, E2E) when the concurrency is N as $L_{T,N}$. By increasing N, $L_{T,N}$ also increases due to the additional workload. To reduce the inherent differences in the latencies of different components, we worked with the normalized latency: $R_{T,N} = \frac{L_{T,N}}{L_{T,1}}$.

We define *bottlenecks* as measured latencies that indicate the corresponding services reach maximum capacities faster than the others as N grows. If we plot $\langle N, R_{T,N} \rangle$ curves on a $\langle log_2(N), log_2(R_{T,N}) \rangle$ plot, bottlenecks will reveal increasingly higher $R_{T,N}$ values as N increases. Therefore, we focus on curves that overwhelm others starting from a certain N.



(a) Before optimization, the bottleneck lies in request handlers (those of Req_{C1} and Req_{C2})



(b) After optimization, the bottleneck has moved from request handlers to the network.

Figure 6: Tuning Results. The bottleneck moved from services (Svc_C) to the network (RT).

As described in Figure 6a, curves of Req_{C1}, Req_{C2} and $E2E$ overwhelmed the others when $N \geq 4$. Since $E2E$ is an overall performance indicator, optimizations should firstly be applied to Svc_C, the processor of Req_{C1} and Req_{C2}, and only after that, to other services. The optimizations we made can be briefly described in three parts:

Optimize Req_{C2} handler. Originally, Req_{C2} handler converted the default currency (USD) of the product to user's currency by first converting to EUR, and then from EUR to user currency. We made Req_{C2} handler convert default currency directly to user currency.

Optimize Svc_C's upstream service. We optimized the logic of Svc_{FE} to reduce the number of Req_{C2}s, and thus lowered the workload of Svc_C. When Svc_{FE} converts currencies for

multiple products, it gets the current exchange rate by sending only one Req_{C2} to Svc_C, and converts the currencies locally.

Optimize the other request handlers. We discovered that Req_{P1} handler occupied too many disk I/O resources. Instead of using a database or cache layers, Req_{P1} read product list from JSON files. We implemented a cache layer to prevent Svc_P from reading JSON files, leading to a fewer disk I/Os.

The results of the first round of tuning are shown in Figure 6. The bottleneck flipped from the request handlers to the network, and the throughput increased from 579 op/s to 1278 op/s on Cluster2.

The request handlers were far from reaching their maximum capacities while the network resource exhausted and made *RT* the bottleneck. After applying more rounds of optimizations, the throughput of **Cluster2** finally reached 2122 op/s.

5.4.2 Root Cause Analyses. Note that fault injection frameworks aim to identify that bugs exists, but do little to help isolate or fix bugs. The following story tells how we localized bugs with *3mb-payloads* collected from fault injection, showing that the advantage of coupling tracing and fault injection makes 3MileBeach's traces can be used to localize bugs found by 3MileBeach's fault injection.

Exhaustive RLFI and DeepRLFI Bug. The DeepRLFI bug can be reached by crashing Svc_A and Svc_C, which might make an intermediate service, namely Svc_I, crash or return an unexpected response to Svc_{FE}. However, if triggering faults involving Svc_I could also fail Svc_{FE}, we should have found certain *3mb-payloads* contain Svc_I. As shown in Figure 5c, the root cause of this fault is a DeepRLFI bug that makes Svc_{FE} unable to handle the situation when Svc_A and Svc_C crash.

Targeted-Exhaustive TFI and SimpleTFI Bug. As illustrated in Figure 5d, the SimpleTFI bug, a TOCTTOU bug, can be reached when Svc_{FE} gets the proper response from the first invocation of Svc_C (TOC), but does not check the status of the rest requests (TOU). 3MileBeach is able to locate this fault by applying TFI, enabling users to perform experiments in ways that existing tools can not cover, or cannot handle efficiently.

6 CONCLUSION AND FUTURE WORK

While 3MileBeach provides a rich set of *mechanisms* for fault injection, we were careful to decouple this from the problem of the *strategy* for choosing experiments. In many cases, our lightweight fault simulation capability makes it possible to *exhaustively* search space of experiments. 3MileBeach is intended to augment existing *integration testing* approaches, ensuring that fault tolerance guarantees and expectations are upheld not merely locally (microservice by microservice), but under composition. 3MileBeach can also be applied earlier in the software life cycle.

We presented 3MileBeach, a tracer with teeth. 3MileBeach's design circumvents the common instrumentation costs required to realize tracing and fault injection capabilities by interposing on boundary components. This interposition allows us to make guarantees about trace completeness and offer fault injection *precision* that exceeds that of state-of-the-art proprietary solutions. Another strength is that fault simulation brings no blast radius, allowing fault injection in production environment. Perhaps most importantly, we showed how 3MileBeach provides a platform for innovation in the space of problem localization.

Future Work: For more complicated applications that involve more services, more pairwise communication, and more discrete temporal intervals, exhaustive searches will be intractable. We plan to integrate 3MileBeach with experiment selection technologies such as LDFI [20, 21]. A barrier to entry for LDFI and related techniques is that they assume the existence of fine-grained tracing and fault injection mechanisms. 3MileBeach readily satisfies this assumption. To take advantage of TFI, LDFI will need to be extended to reason about the temporal dimension in traces explicitly.

We also plan to investigate the use of 3MileBeach as an adjunct to dependency injection techniques during the unit testing of individual services and integration with step-oriented debuggers.

This work mostly focused on data flows (*i.e.*, DRC and SRC) that can provide immediate responses. This scenario is generic among applications that require instant human-computer interactions such as searching engine, social media, online shopping, *etc.* On the other hand, 3MileBeach can be extended to support ***A**synchronous **R**equest-response **C**ircle* (**ARC**), which is popular among systems that rely on *non-blocking* communications and distributed coordination. Typical examples are data consistency approaches (*e.g.*, Zookeeper [41], ETCD [5], *etc.*) and distributed databases (*e.g.*, HBase [39]).

ACKNOWLEDGEMENT

This research is supported by National Science Foundation (Award #1652368) and by gifts from Facebook and Ebay. The authors wish to thank the shepherd, Raja Sambasivan, and the anonymous reviewers for their helpful feedback which materially improved this submission, Aria Diamond, Arjun Loganathan, Qitong Wang, and Ruoyu Su for their fruitful discussion and feedback.

REFERENCES

[1] [n.d.]. AWS X-Ray Developer Guide. https://docs.aws.amazon.com/xray/latest/devguide/xray-guide.pdf.

[2] [n.d.]. Chaos Community Broadcast. http://chaos.community.

[3] [n.d.]. Chaos Toolkit. https://chaostoolkit.org/.

[4] [n.d.]. Docker Command Line Reference. https://docs.docker.com/engine/reference/commandline/docker/.

[5] [n.d.]. etcd, A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io.

[6] [n.d.]. GKE overview. https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview.

[7] [n.d.]. Gorilla, the golang web toolkit. https://www.gorillatoolkit.org.

[8] [n.d.]. Gremlin. https://www.gremlin.com/.

[9] [n.d.]. gRPC. https://grpc.io.

[10] [n.d.]. Hipster Shop. https://github.com/census-ecosystem/opencensus-microservices-demo.

[11] [n.d.]. Introducing JSON. https://www.json.org/json-en.html.

[12] [n.d.]. JSON and serialization. https://flutter.dev/docs/development/data-and-backend/json.

[13] [n.d.]. Kubectl Reference Docs. https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands.

[14] [n.d.]. OpenTracing. https://opentracing.io/.

[15] [n.d.]. Protocol Buffers. https://developers.google.com/protocol-buffers.

[16] 2017. ChAP: Chaos Automation Platform. https://netflixtechblog.com/chap-chaos-automation-platform-53e6d528371f.

[17] R. Abbott, J. Chin, Jed Donnelley, W. Konigsford, S. Tokubo, and D. Webb. 1976. Security Analysis and Enhancements of Computer Operating Systems. (Apr. 1976), 70.

[18] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454

[19] James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (Nov. 1983), 832–843. https://doi.org/10.1145/182.358434

[20] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 17–28. https://doi.org/10.1145/2987550.2987555

[21] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-Driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[22] Kolton Andrus. 2018. The Next Step: Application-Level Fault Injection. https://www.gremlin.com/blog/the-next-step-application-level-fault-injection.

[23] Kolton Andrus, Naresh Gopalani, and Ben Schmaus. 2014. FIT : Failure Injection Testing. http://techblog.netflix.com/2014/10/fit-failure-injection-testing.html.

[24] Chris Aniszczyk. 2012. Distributed Systems Tracing with Zipkin. https://blog.twitter.com/2012/distributed-systems-tracing-with-zipkin.

[25] Jakob Axelsson and Mats Skoglund. 2016. Quality assurance in software ecosystems: A systematic literature mapping and research agenda. *Journal of Systems and Software* 114 (2016), 69–81. https://doi.org/10.1016/j.jss.2015.12.020

[26] Peter Bailis, Peter Alvaro, and Sumit Gulwani. 2017. Research for Practice: Tracing and Debugging Distributed Systems; Programming by Examples. *Commun. ACM* 60, 7 (June 2017), 46–49. https://doi.org/10.1145/3052942

[27] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (May 2016), 35–41. https://doi.org/10.1109/MS.2016.60

[28] Ivan Beschastnikh, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2016. Debugging distributed systems: Challenges and options for validation and debugging. *Commun. ACM* 59, 8 (Aug. 2016), 32–37.

[29] Matt Bishop. 1995. Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux.

[30] Matt Bishop and Michael Dilger. 1996. Checking for Race Conditions in File Accesses. *Computing Systems* 9, 2 (Spring 1996), 131–152.

[31] Daniel Bittman, Ethan L. Miller, and Peter Alvaro. 2019. Co-evolving Tracing and Fault Injection with Box of Pain. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/bittman

[32] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 217–231. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chow

[33] Nicos Christofides. 1975. *Graph Theory: An Algorithmic Approach (Computer Science and Applied Mathematics)*. Academic Press, Inc., USA. 170–174 pages.

[34] J. A. Clark and D. K. Pradhan. 1995. Fault injection: a method for validating computer-system dependability. *Computer* 28, 6 (1995), 47–56. https://doi.org/10.1109/2.386985

[35] R. D'Andrea, C. Beck, and G. E. Dullerud. 1999. Temporal discretization of spatially distributed systems. In *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*, Vol. 1. 197–202. https://doi.org/10.1109/CDC.1999.832774

[36] S. Dawson, F. Jahanian, and T. Mitton. 1996. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In *Proceedings of IEEE International Computer Performance and Dependability Symposium*. 56–. https://doi.org/10.1109/IPDS.1996.540200

[37] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation.

[38] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[39] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.".

[40] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and Enhancing In Situ System Observability for Failure Detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 1–16. https://www.usenix.org/conference/osdi18/presentation/huang

[41] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[42] Yury Izrailevsky and Ariel Tseitlin. 2011. The Netflix Simian Army. http://techblog.netflix.com/2011/07/netflix-simian-army.html.

[43] K. Jain and R. Sekar. 1999. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *In Proc. Network and Distributed Systems Security Symposium*.

[44] Michael B. Jones. 1993. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 80–93. https://doi.org/10.1145/168619.168626

[45] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 34–50. https://doi.org/10.1145/3132747.3132749

[46] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. 1995. FERRARI: a flexible software-based fault and error injection system. *IEEE Trans. Comput.* 44, 2 (1995), 248–260. https://doi.org/10.1109/12.364536

[47] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[48] Pedro Henrique B. Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019*.

[49] Jonathan Mace and Rodrigo Fonseca. 2018. Universal Context Propagation for Distributed System Instrumentation. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 8, 18 pages. https://doi.org/10.1145/3190508.3190526

[50] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2016. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/atc16/technical-sessions/presentation/mace

[51] P. D. Marinescu and G. Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. 379–388. https://doi.org/10.1109/DSN.2009.5270313

[52] Keith Ansel Marzullo. 1984. *Maintaining the Time in a Distributed System: An Example of a Loosely-Coupled Distributed Service (Synchronization, Fault-Tolerance, Debugging)*. Ph.D. Dissertation. Stanford, CA, USA. AAI8506272.

[53] Ashutosh Raina and Ramprasad Ellupuru. 2019. Madaari: Ordering for the Monkeys. USENIX Association, Brooklyn, NY.

[54] Emily Reinhold. 2016. Rewriting Uber Engineering. https://eng.uber.com/building-tincup/.

[55] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. 2006. WAP5: Black-Box Performance Debugging for Wide-Area Systems. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland) *(WWW '06)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/1135777.1135830

[56] C. Rosenthal and N. Jones. 2020. *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media.

[57] Raja Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory Ganger. 2014. So, you want to trace your distributed system? Key design insights from years of practical experience. (Apr. 2014).

[58] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[59] M. Shahin, M. Ali Babar, and L. Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access* 5 (2017), 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

[60] Arjun Shenoy. 2016. A Deep Dive into Simoorg: Our Open Source Failure Induction Framework. https://engineering.linkedin.com/blog/2016/03/deep-dive-Simoorg-open-source-failure-induction-framework.

[61] Yuri Shkuro. 2017. Evolving Distributed Tracing at Uber Engineering. https://eng.uber.com/distributed-tracing/.

[62] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. http://research.google.com/archive/papers/dapper-2010-1.pdf

[63] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press, USA.

[64] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. 2004. Constructing Services with Interposable Virtual Hardware. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (San Francisco, California) *(NSDI'04)*. USENIX Association, USA, 13.

[65] 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* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 333–346. https://doi.org/10.1145/3267809.3267839