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 - Temporal Fault Injection (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:


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 Directed Acyclic Graph (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 Stochastic Fault Injection (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 Principled Fault Injection (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, Request Level Fault Injection (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.

<table>
<thead>
<tr>
<th>Approach</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$C_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFI</td>
<td>×</td>
<td>×</td>
<td>✓$^1$</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>PFI</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RLFI</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×$^2$</td>
</tr>
<tr>
<td>TFI</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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) is only available as a proprietary technology. Unfortu-

3MileBeach: A Tracer with Teeth SoCC ’21, November 1–4, 2021, Seattle, WA, USA

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.

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

23MileBeach 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 PRODUC to calculate the total purchasing price, and CURRNCY SERVICE 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 Time Of Check To Time Of Use (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 validation, 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 < t₀);
2) between FRONTEND’s first request and CardHandler’s first response (t₀ ≤ t < t₁);
3) between CardHandler’s first response and FRONTEND’s second request (t₁ ≤ t < t₂);
4) between FRONTEND’s second request and CardHandler’s second response (t₂ ≤ t < t₃).

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 (Req₁ and Req₂) in a client-level request flow.](image)

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 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:

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

\( P_I \) Inbound direction. All functions are assigned to a single thread where the inbound component sends requests to the service handler and the service handler sends responses to the outbound component.

\( P_{I/O} \) Bidirectional direction. All functions are assigned to a single thread where the inbound and outbound components send requests to the service handler and the service handler sends responses to the inbound and outbound components.

\( P_{I/O} \) Concurrent direction. All functions are assigned to a single thread where the inbound and outbound components send requests to the service handler and the service handler sends responses to the inbound and outbound components concurrently.
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, Direct Response Circle (DRC) and Synchronized Request-Response Circle (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.).

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 fault injection configuration-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 UIDs.

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 (Svc1) when it sends a request (Req1) at $t_1$ and receives the corresponding response (Rsp1) at $t_4$, and the other two events are recorded by the responder (Svc2) at $t_2$ and $t_3$.

4.2.2 Fault Injection Configuration. We use fault injection configuration (FIC) to describe the objective of a TFI or RFII 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 $Svc_0$, $Svc_1$, $Svc_2$, $Svc_{n-1}$.

RFII injects faults to services prior to the life cycle of a client-level request. We use FIC {Type: Crash, Name: $Svc_i$}
We use services will not be invoked, we might waste time per-
\[\text{val (1)}\].
\[\text{Req}_{1}\]
\[\text{Req}_{2}\]
\[\text{Crash}, \text{Name: Req}_1, \text{Times: 1}\]]. 3MileBeach does not assign the entire trace 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 \textsc{Deserialize} and \textsc{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 \textsc{Deserialize}' and \textsc{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 \textit{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).**

- Framework RECVs a REQUEST from an upstream service or the client;
- Inbound component invokes \textsc{Deserialize}' which calls \textsc{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);
- 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetUUID</td>
<td>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 requestor; 2) The Rest. The uuid can be fetched from trace.Events recorded by the other service that participates in the same SRC.</td>
</tr>
<tr>
<td>trace.Extend</td>
<td>Extends the input event to trace.Events and updates trace.FICs.</td>
</tr>
<tr>
<td>S.SetTrace</td>
<td>Gets trace by ID, calls trace.Extend(event), and eventually returns trace.</td>
</tr>
<tr>
<td>S.SetTrace</td>
<td>Upserts trace according to trace.ID. More specifically, during updates, S merges the Events from both traces and recalculates trace.FICs.</td>
</tr>
</tbody>
</table>

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 = CURRENT_TIMESTAMP()
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 = CURRENT_TIMESTAMP()
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

delays trace (#L11), and calls Serialize (#L16) to serialize the RESPONSE;

⇒ Framework SENDs the RESPONSE back to the upstream service or the client.

Synchronized Request-Response Circle (SRC).

→ Service handler SENDs a REQUEST to the downstream service through a blocked function call;

→ 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;

⇒ Framework SENDs the REQUEST to the downstream service and waits for the RESPONSE;

→ Framework RECVs the RESPONSE;

⇒ 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);

⇒ 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 SimulateFault 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 End-To-End (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 (SvcFE), Cart (SvcCart), Recommendation, ProductCatalog (SvcP), Shipping, Currency (SvcC), Payment, Email, Checkout, and Ad (SvcA). 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 SvcFE 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 SvcFE.

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 SvcFE. To satisfy a client request, SvcFE generates calls to four downstream services that will respond to five GRPC requests including SvcA’s GetAds (ReqA), SvcCart’s GetCart, SvcC’s GetCurrencies (ReqC1) and CurrencyConversion (ReqC2), and SvcP’s ListProduct (ReqP).

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 SvcCart, to get an apples-to-apples comparison, 3MileBeach will not collect events from SvcCart, either. However, 3MileBeach is still able to simulate faults to SvcCart due to the interposition point which is located at SvcFE’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.

<table>
<thead>
<tr>
<th>Table 3: Cluster Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>Cluster1</td>
</tr>
<tr>
<td>Cluster2</td>
</tr>
</tbody>
</table>

3MileBeach: A Tracer with Teeth

SoCC ’21, November 1–4, 2021, Seattle, WA, USA
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.

<table>
<thead>
<tr>
<th>Sample Rate</th>
<th>Overhead</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>19.9%</td>
<td>46.3%</td>
</tr>
<tr>
<td>1/4</td>
<td>7.45%</td>
<td>18.4%</td>
</tr>
<tr>
<td>1/16</td>
<td>6.97%</td>
<td>14.1%</td>
</tr>
<tr>
<td>1/64</td>
<td>5.33%</td>
<td>7.34%</td>
</tr>
</tbody>
</table>

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 SocFE:

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

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

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

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

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

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 $S_{FE}$ 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. \( t_{\text{start}} = \text{now}() \)
3. for each \( fic \) in \( fics \) do
4. \hspace{1em} Shut down service \( fic \).Name
5. Client sends a request, and records status code
6. for each \( fic \) in \( fics \) do
7. \hspace{1em} Restart service \( fic \).Name
8. if Services fail to restart, or errors happen in other services then
9. \hspace{1em} Redeploy the application
10. Wait until all services are ready
11. \( t_{\text{end}} = \text{now}() \)

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Cases</th>
<th>3MILEBEACH</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-E RLFI</td>
<td>16</td>
<td>200ms</td>
<td>9m34s</td>
</tr>
<tr>
<td>E RLFI</td>
<td>512</td>
<td>5s</td>
<td>5h21m</td>
</tr>
<tr>
<td>T-E TFI</td>
<td>16384</td>
<td>90s</td>
<td>N/A</td>
</tr>
</tbody>
</table>

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 re-running 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 Round-Trip (RT) latency between the client and SucFE, and the processing latencies measured by request handlers ReqA1, ReqC1, ReqC2, and ReqP1. We denote the average measured latency of type \( T \) (e.g., RT, E2E) when the concurrency is \( N \) as \( \ell_T.N \). By increasing \( N \), \( \ell_T.N \) also increases due to the additional workload. To reduce the inherent differences in the latencies of different components, we worked with the normalized latency: \( R_T.N = \frac{\ell_T.N}{\ell_T} \).

We define bottlenecks as measured latencies that indicate the corresponding services reach maximum capacities faster than the others as \( N \) grows. If we plot \( (N, R_T.N) \) curves on a \( (\log_2(N), \log_2(R_T.N)) \) plot, bottlenecks will reveal increasingly higher \( R_T.N \) values as \( N \) increases. Therefore, we focus on curves that overwhelmed others starting from a certain \( N \).

As described in Figure 6a, curves of \( ReqC1 \), \( ReqC2 \) and \( E2E \) overwhelmed the others when \( N \geq 4 \). Since \( E2E \) is an overall performance indicator, optimizations should firstly be applied to \( SucC \), the processor of \( ReqC1 \) and \( ReqC2 \), and only after that, to other services. The optimizations we made can be briefly described in three parts:

Optimize \( ReqC2 \) handler. Originally, \( ReqC2 \) 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 \( ReqC2 \) handler convert default currency directly to user currency.

Optimize \( SucC \)’s upstream service. We optimized the logic of \( SucFE \) to reduce the number of \( ReqC2 \)s, and thus lowered the workload of \( SucC \). When \( SucFE \) converts currencies for
While 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 Asynchronous Request-response Circle (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.
Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using C.


