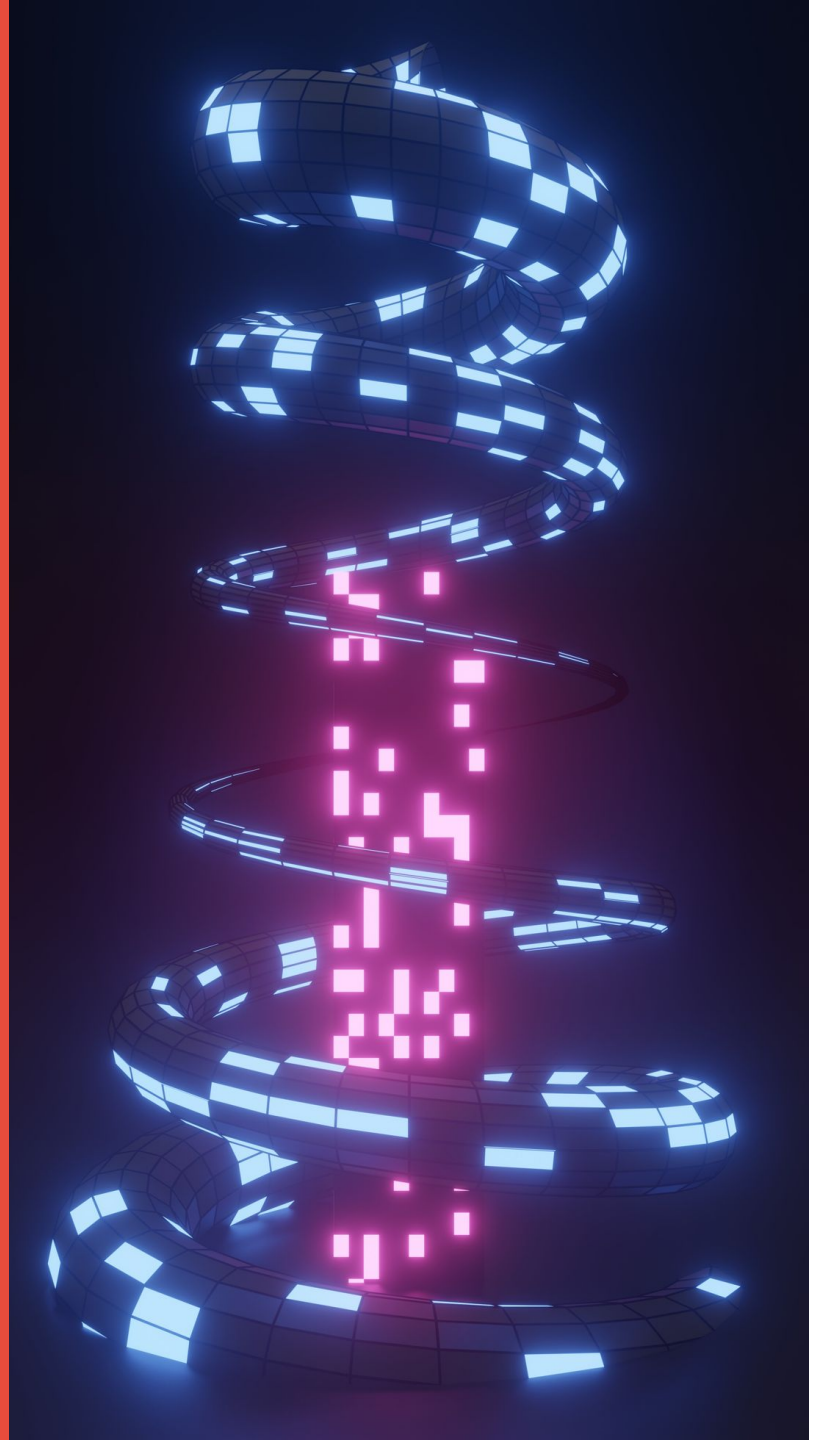


Not Your Grandma's Distributed Programming

Heather Miller

Carnegie Mellon University

Everything is now distributed.

Maybe you don’t think about it that way, but if you are making requests to external APIs, you are working with a distributed system.

If you make an HTTP request somewhere, you're writing a distributed program.

Therefore, you might do a lot of distributed programming, even if you don't think of yourself as a hardcore distributed systems engineer.

what are things you need to think about when you are building a distributed system?

throw some ideas out, whatever comes to your mind

• Concurrency

Concurrent processes need to work together in order to achieve the goals of the system they comprise.

A concurrent system may be:

- an operating system,
- a subsystem within an operating system, or
- an application-level service running above an operating system.

In all cases, a concurrent system may be distributed across a number of computers rather than centralized in a single computer.

• Partial Failure

the unavailability of one or more dependent systems/services/processes can render an entire application unusable.

Partial failures are difficult to deal with. Fundamentally, this is because systems/services/processes may be uncertain about the state of failed ones. Operational ones may become blocked, unable to complete their task, until this uncertainty is resolved.

(in contrast, a total failure is when all systems/services/process are down.)

Some Answers

a quick aside on concurrency

a quick aside on
partial failure

than objects. Most programmers use pointers as references for many different kinds of entities. These pointers must either be replaced with something that can be used in cross-address space calls or the programmer will need to be aware of the difference between such calls (which will either not allow pointers to such entities, or do something special with those pointers) and local calls. Again, while this could be done, it does violate the doctrine of complete unity between local and remote calls. Because of memory access constraints, the two *have* to differ.

The danger lies in promoting the myth that "remote access and local access are exactly the same" and not enforcing the myth. An underlying mechanism that does not unify all memory accesses while still promoting this myth is both misleading and prone to error. Programmers buying into the myth may believe that they do not have to change the way they think about programming. The programmer is therefore quite likely to make the mistake of using a pointer in the wrong context, producing incorrect results. "Remote is just like local," such programmers think, "so we have just one unified programming model." Seemingly, programmers need not change their style of programming. In an incomplete implementation of the underlying mechanism, or one that allows an implementation language that in turn allows direct access to local memory, the system does not take care of all memory accesses, and errors are bound to occur. These errors occur because the programmer is not aware of the difference between local and remote access and what is actually happening "under the covers."

The alternative is to explain the difference between local and remote access, making the programmer aware that remote address space access is very different from local access. Even if some of the pain is taken away by using an interface definition language like that specified in [1] and having it generate an intelligent language mapping for operation invocation on distributed objects, the programmer aware of the difference will not make the mistake of using pointers for cross-address space access. The programmer will know it is incorrect. By not masking the difference, the programmer is able to learn when to use one method of access and when to use the other.

Just as with latency, it is logically possible that the difference between local and remote memory access could be completely papered over and a single model of both presented to the programmer. When we turn to the problems introduced to distributed computing by partial failure and concurrency, however, it is not clear that such a unification is even conceptually possible.

4.3 Partial failure and concurrency

While unlikely, it is at least logically possible that the differences in latency and memory access between local computing and distributed computing could be masked. It is not clear that such a masking could be done in such a way that the local computing paradigm could be used to produce distributed applications, but it might still be possible to allow some new programming technique to be used for both activities. Such a masking does not even seem to be logically possible, however, in the case of partial failure and concurrency. These aspects appear to be different in kind in the case of distributed and local computing.[2]

Partial failure is a central reality of distributed computing. Both the local and the distributed world contain components that are subject to periodic failure. In the case of local computing, such failures are either total, affecting all of the entities that are working together in an application, or detectable by some central resource allocator (such as the operating system on the local machine).

This is not the case in distributed computing, where one component (machine, network link) can fail while the others continue. Not only is the failure of the distributed components independent, but there is no common agent that is able to determine what component has failed and inform the other components of that failure, no global state that can be examined that allows determination of exactly what error has occurred. In a distributed system, the failure of a network link is indistinguishable from the failure of a processor on the other side of that link.

These sorts of failures are not the same as mere exception raising or the inability to complete a task, which can occur in the case of local computing. This type of failure is caused when a machine crashes during the execution of an object invocation or a network link goes down, occurrences that cause the target object to simply disappear rather than return control to the caller. A central problem in distributed computing is insuring that the state of the

2. In fact, authors such as Schroeder [12] and Hadzilacos and Toueg [13] take partial failure and concurrency to be the defining problems of distributed computing.

WALDO ET AL, 1994

Note on Distributed Computing

Partial failure is described nicely in Waldo et al's 1994 paper...

In particular, it contrasts what partial failure means in a single-machine versus distributed scenario.

whole system is consistent after such a failure; this is a problem that simply does not occur in local computing.

The reality of partial failure has a profound effect on how one designs interfaces and on the semantics of the operations in an interface. Partial failure requires that programs deal with indeterminacy. When a local component fails, it is possible to know the state of the system that caused the failure and the state of the system after the failure. No such determination can be made in the case of a distributed system. Instead, the interfaces that are used for the communication must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial failures.

Being robust in the face of partial failure requires some expression at the interface level. Merely improving the implementation of one component is not sufficient. The interfaces that connect the components must be able to state whenever possible the cause of failure, and there must be interfaces that allow reconstruction of a reasonable state when failure occurs and the cause cannot be determined.

If an object is coresident in an address space with its caller, partial failure is not possible. A function may not complete normally, but it always completes. There is no indeterminism about how much of the computation completed. Partial completion can occur only as a result of circumstances that will cause the other components to fail.

The addition of partial failure as a possibility in the case of distributed computing does not mean that a single object model cannot be used for both distributed computing and local computing. The question is not "can you make remote method invocation look like local method invocation?" but rather "what is the price of making remote method invocation identical to local method invocation?" One of two paths must be chosen if one is going to have a unified model.

The first path is to treat all objects as if they were local and design all interfaces as if the objects calling them, and being called by them, were local. The result of choosing this path is that the resulting model, when used to produce distributed systems, is essentially indeterministic in the face of partial failure and consequently fragile and nonrobust. This path essentially requires ignoring the extra failure modes of distributed computing. Since one can't get rid of those failures, the price of adopting the model is to require that such failures are unhandled and catastrophic.

The other path is to design all interfaces as if they were remote. That is, the semantics and operations are all designed to be deterministic in the face of failure, both total and partial. However, this introduces unnecessary guarantees and semantics for objects that are never intended to be used remotely. Like the approach to memory access that attempts to require that all access is through system-defined references instead of pointers, this approach must also either rely on the discipline of the programmers using the system or change the implementation language so that all of the forms of distributed indeterminacy are forced to be dealt with on all object invocations.

This approach would also defeat the overall purpose of unifying the object models. The real reason for attempting such a unification is to make distributed computing more like local computing and thus make distributed computing easier. This second approach to unifying the models makes local computing as complex as distributed computing. Rather than encouraging the production of distributed applications, such a model will discourage its own adoption by making all object-based computing more difficult.

Similar arguments hold for concurrency. Distributed objects by their nature must handle concurrent method invocations. The same dichotomy applies if one insists on a unified programming model. Either all objects must bear the weight of concurrency semantics, or all objects must ignore the problem and hope for the best when distributed. Again, this is an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. So either the overall programming model must ignore significant modes of failure, resulting in a fragile system; or the overall programming model must assume a worst-case complexity model for all objects within a program, making the production of any program, distributed or not, more difficult.

One might argue that a multi-threaded application needs to deal with these same issues. However, there is a subtle difference. In a multi-threaded application, there is no real source of indeterminacy of invocations of operations. The application programmer has complete control over invocation order when desired. A distributed system by its nature

WALDO ET AL, 1994

Note on Distributed Computing

Partial failure is described nicely in Waldo et al's 1994 paper...

In particular, it contrasts what partial failure means in a single-machine versus distributed scenario.

• Concurrency

Concurrent processes need to work together in order to achieve the goals of the system they comprise.

A concurrent system may be:

- an operating system,
- a subsystem within an operating system, or
- an application-level service running above an operating system.

In all cases, a concurrent system may be distributed across a number of computers rather than centralized in a single computer.

• Partial Failure

the unavailability of one or more dependent systems/services/processes can render an entire application unusable.

Partial failures are difficult to deal with. Fundamentally, this is because systems/services/processes may be uncertain about the state of failed ones. Operational ones may become blocked, unable to complete their task, until this uncertainty is resolved.

(in contrast, a total failure is when all systems/services/process are down.)

Some Answers

when you sit down to debug your request, are you typically worrying about concurrency and partial failure?

be honest

NO

(it's okay, you shouldn't have to.)

What are examples of distributed programming people do every day?

Microservices

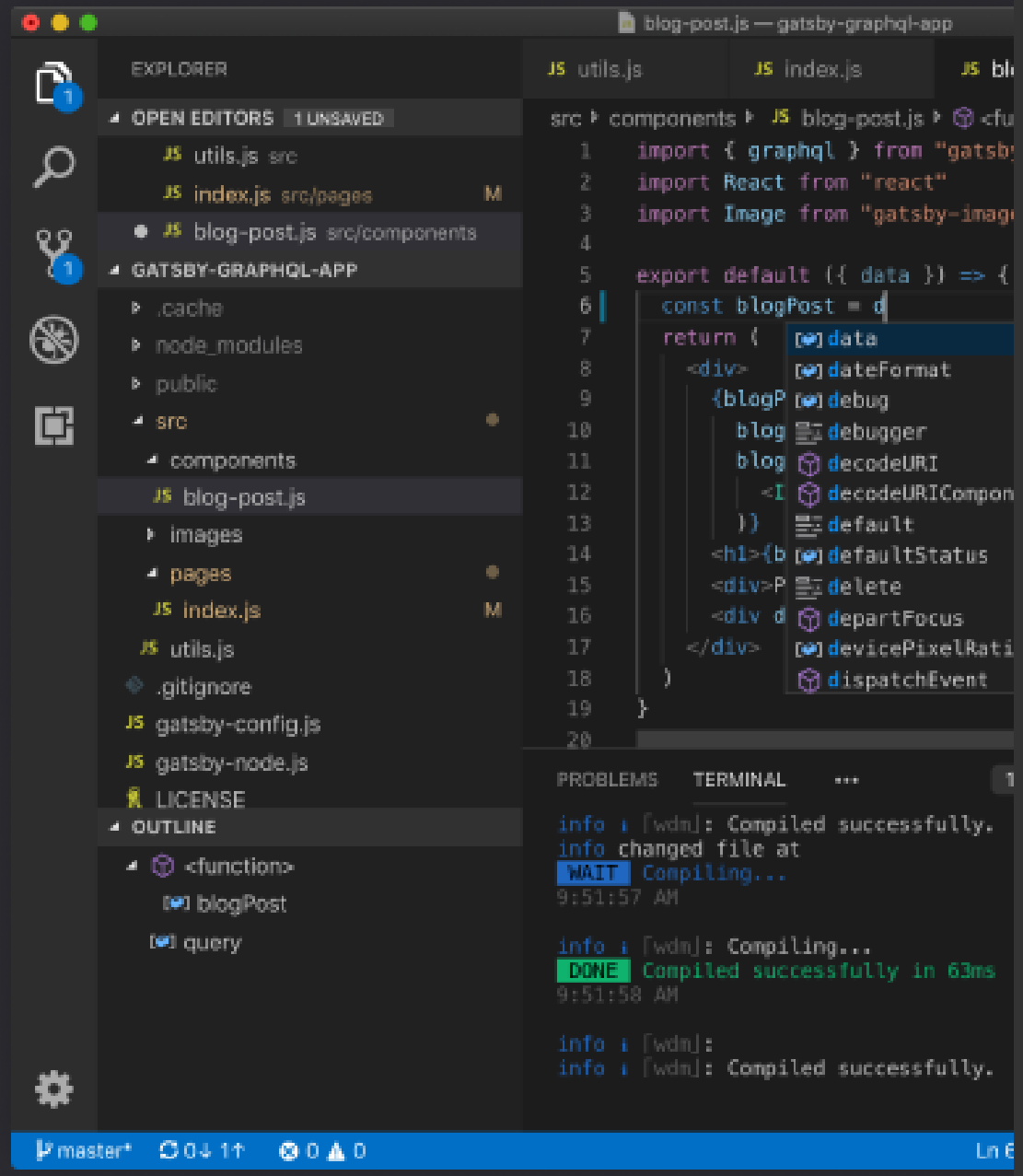
Microservices are teams of people organized around a component with a singular responsibility. That team of people write that component in whatever language they want and publish an internal API for it for teams of others in the org to make requests to in their own components.

OpenAPIs

Think about frontend development. Many frontend developers are given API endpoints and instructed to pull from them and create an interactive UX. What's more, UIs tend to be extremely complex state machines integrating stateful actions from user interaction. Frontend developers are doing distributed programming *with* the human-in-the-loop.

Lots of stuff that's ...aS

Lots of “as-a-service” companies offer APIs that do some kind of work for you. For example, you might send an image and get back textual annotations of what might be contained in the image. What happens if that API that you're using changes, changes behavior, is shut down, etc?



Remember when you just needed Java, a good IDE, and some tests?

That same experience should exist in today's world of software development.

These are things that our tools and frameworks should help us with.

Thesis argument for this entire talk...

Better Tooling

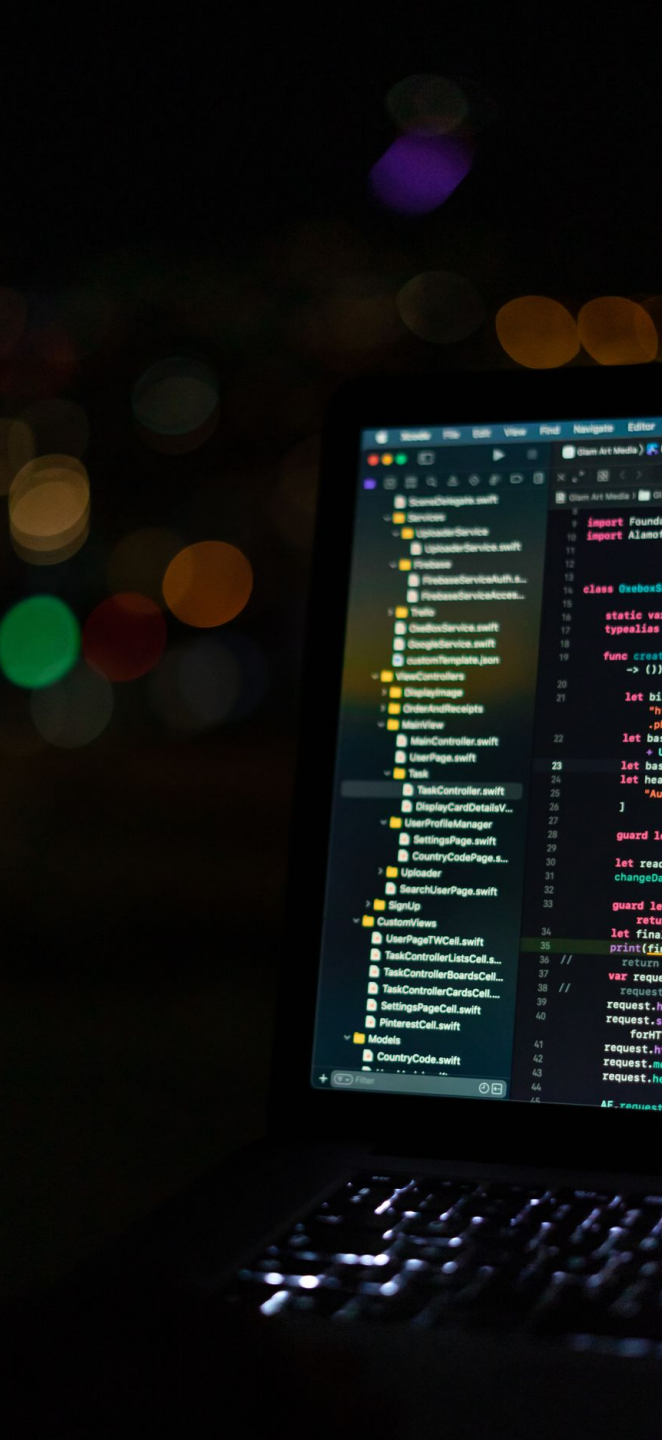
- Where are our debuggers, autocomplete, and testing frameworks for these things!?

Better Abstractions

- Abstractions should always make the Correct Thing(TM) the path of least resistance.
- **The programming abstraction that you are using should guide you to do the right thing, whatever that is, without adding extra complexity to your business logic.**

The Future of Distributed Programming Needs Better....

Tooling



Find errors before you ship.

See your whole application.

Things we have in a “regular” single-machine programming environment that we should have in this brave new world of distribution everywhere...

See new functionality.

Debug with a debugger.

Distributed Analogues to: Testing, Debugging, “Autocomplete”

So how close are we to having this development experience? What kind of approaches are out there that can take us a step closer to this vision?

Testing

Tools like Filibuster. Figure out how your program behaves when other programs that it calls misbehave. Do you crash completely? What fallback behaviors are in place? Do those fallback behaviors even work?

Debugging

Someone please invent one! Right now, there are disparate services that together could be used to debug... For example, Lightstep for distributed tracing (kind of like having a stack? Sort of?)

“Autocomplete”

Tools like Akita. A tool should keep track of what other “functions” (yes, yes, APIs) exist that are available to you. Also, do they change?

Filibuster

FILIBUSTER

project site

Testing for applications that are distributed. (e.g., *microservices*.)

Service-level Fault Injection Testing is a technique for identifying issues between distributed components, *during development*, before code ships to production.

Filibuster has been designed to be easy to use, lightweight, and able to be integrated into a continuous integration environment, like GitHub Actions or Amazon's CodeBuild CI/CD environment.

Chris Meiklejohn, CMU

Basic idea:

Automatically generate mocks at points when remote calls are made.

Assuming that we already have functional tests, we know how the software is supposed to behave.

Chris recorded a demo of Filibuster working on a Java application, check it out!

ACM SoCC'21 (paper)

Service-Level Fault Injection Testing

Christopher S. Meiklejohn
Carnegie Mellon University
Pittsburgh, PA, United States
cmeiklej@cs.cmu.edu

Andrea Estrada
Carnegie Mellon University
Pittsburgh, PA, United States
arestrad@andrew.cmu.edu

Yiwen Song
Carnegie Mellon University
Pittsburgh, PA, United States
yiwenson@andrew.cmu.edu

Heather Miller
Carnegie Mellon University
Pittsburgh, PA, United States
heather.miller@cs.cmu.edu

Rohan Padhye
Carnegie Mellon University
Pittsburgh, PA, United States
rohanpadhye@cmu.edu

Abstract

Companies today increasingly rely on microservice architectures to deliver service for their large-scale mobile or web applications. However, not all developers working on these applications are distributed systems engineers and therefore do not anticipate *partial failure*: where one or more of the dependencies of their service might be unavailable once deployed into production. Therefore, it is paramount that these issues be raised early and often, ideally in a testing environment or before the code ships to production.

In this paper, we present an approach called *service-level fault injection testing* and a prototype implementation called Filibuster, that can be used to systematically identify resilience issues early in the development of microservice applications. Filibuster combines static analysis and concolic-style execution with a novel dynamic reduction algorithm to extend existing functional test suites to cover failure scenarios with minimal developer effort. To demonstrate the applicability of our tool, we present a corpus of 4 real-world industrial microservice applications containing bugs. These applications and bugs are taken from publicly available information of chaos engineering experiments run by large companies *in production*. We then demonstrate how all of these chaos experiments could have been run during development instead, and the bugs they discovered detected long before they ended up in production.

CCS Concepts: • **Computer systems organization → Reliability**.

Keywords: fault tolerance, fault injection, verification

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

ACM ISBN 978-1-4503-8638-8/21/11.

ACM Reference Format:
Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. 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/10.1145/3472883.3487005

1 Introduction

Nowadays, large-scale web applications with significant user bases are typically built using a microservice architecture. All companies listed in the Fortune 50 are either hiring for, or have publicly discussed, their use of a microservice architecture to deliver service to their users. The recent popularity of microservice architectures is a direct result of the benefits that this architectural style brings to the organizations that adopt it–smaller teams can focus on individual services written in the best programming language to solve their problem, and are thus able to more rapidly deliver software at scale. However, while microservice architectures help reduce the burden of coordinating changes between teams to the same application, they are known to increase software complexity.

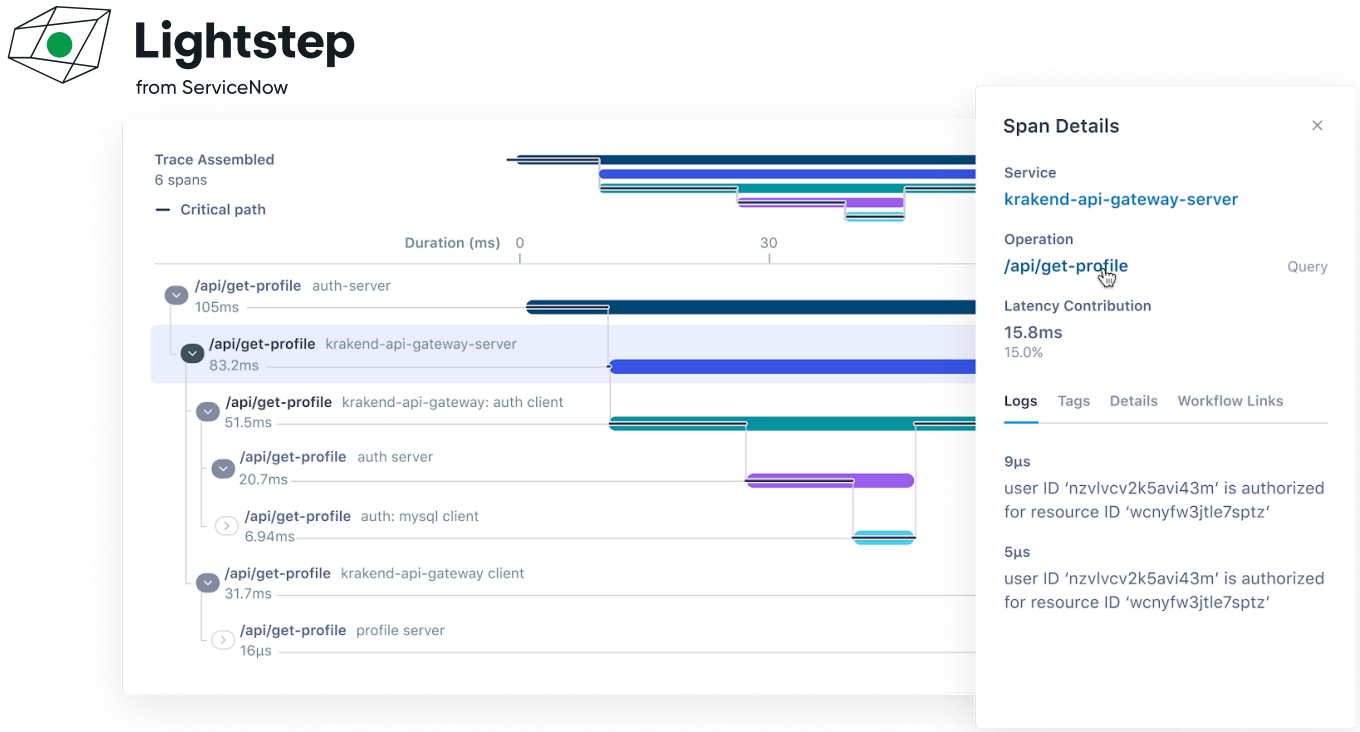
The harsh reality of microservices is that they suddenly force every developer to become a cloud/distributed systems engineer, dealing with the complexity that is inherent in distributed systems [38]. Specifically, *partial failure*, where the unavailability of one or more services can adversely impact the system in unknown ways.

This paper presents the *service-level fault injection testing* (SFIT) technique, as well as Filibuster, a corresponding tool for automatically testing microservice applications for resilience issues related to partial failure. Filibuster can extend an existing functional test suite to cover failure scenarios automatically.

To demonstrate the additional complexity a developer faces when moving from a monolithic architecture to a microservice architecture, consider an application that lets you stream audiobooks. One decomposition [43] of this application into components could look like the following, with one component for each user functionality; storage of audio files, storage of audiobook metadata, storage of user per-

Lightstep

Distributed tracing. Basically, as close as we can get to a stack in a distributed world.

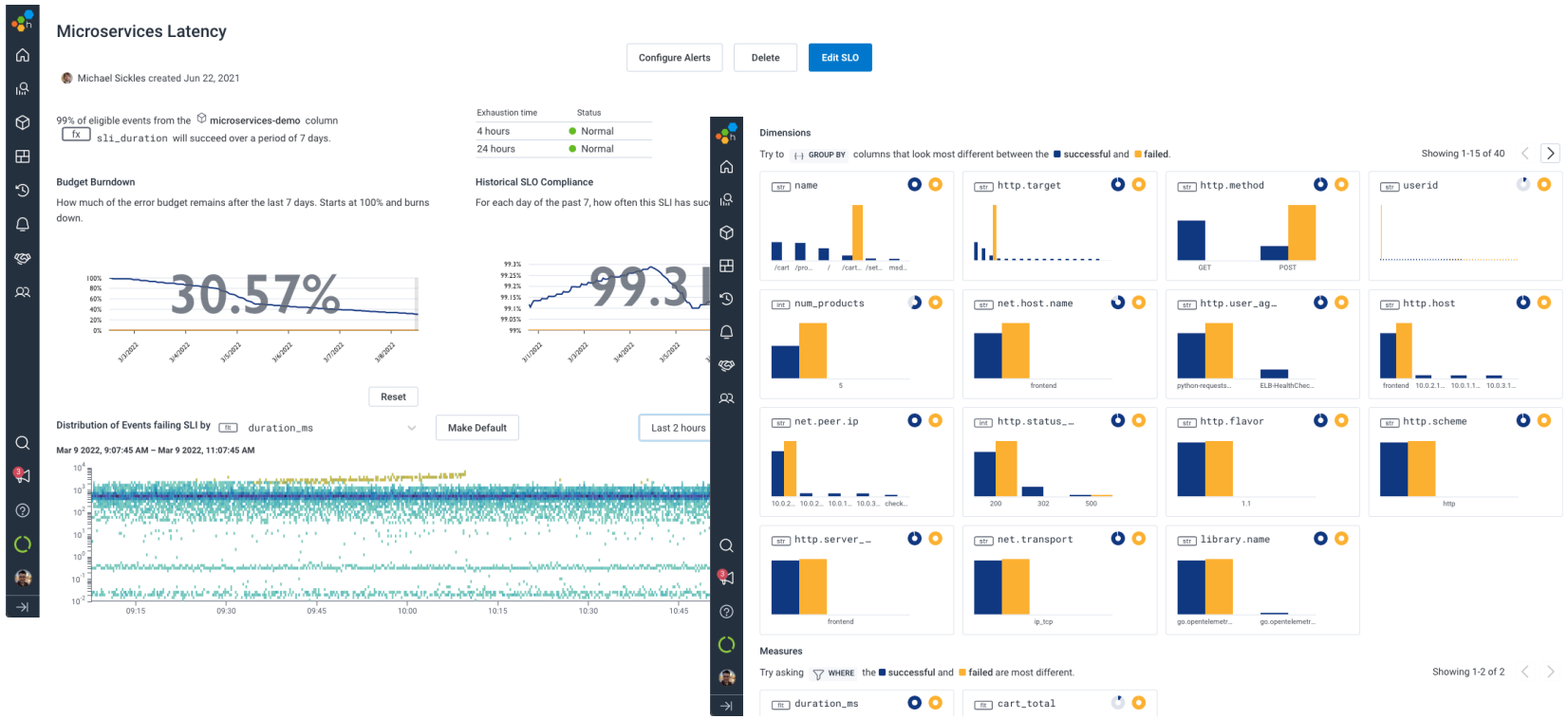


Kind of the beginnings of a debugger, if you consider distributed traces to be something of a stack

Honeycomb

Query all of the info you're collecting, including logs and traces, and try to answer higher-level insights about what's going on between pieces of your application.

honeycomb.io

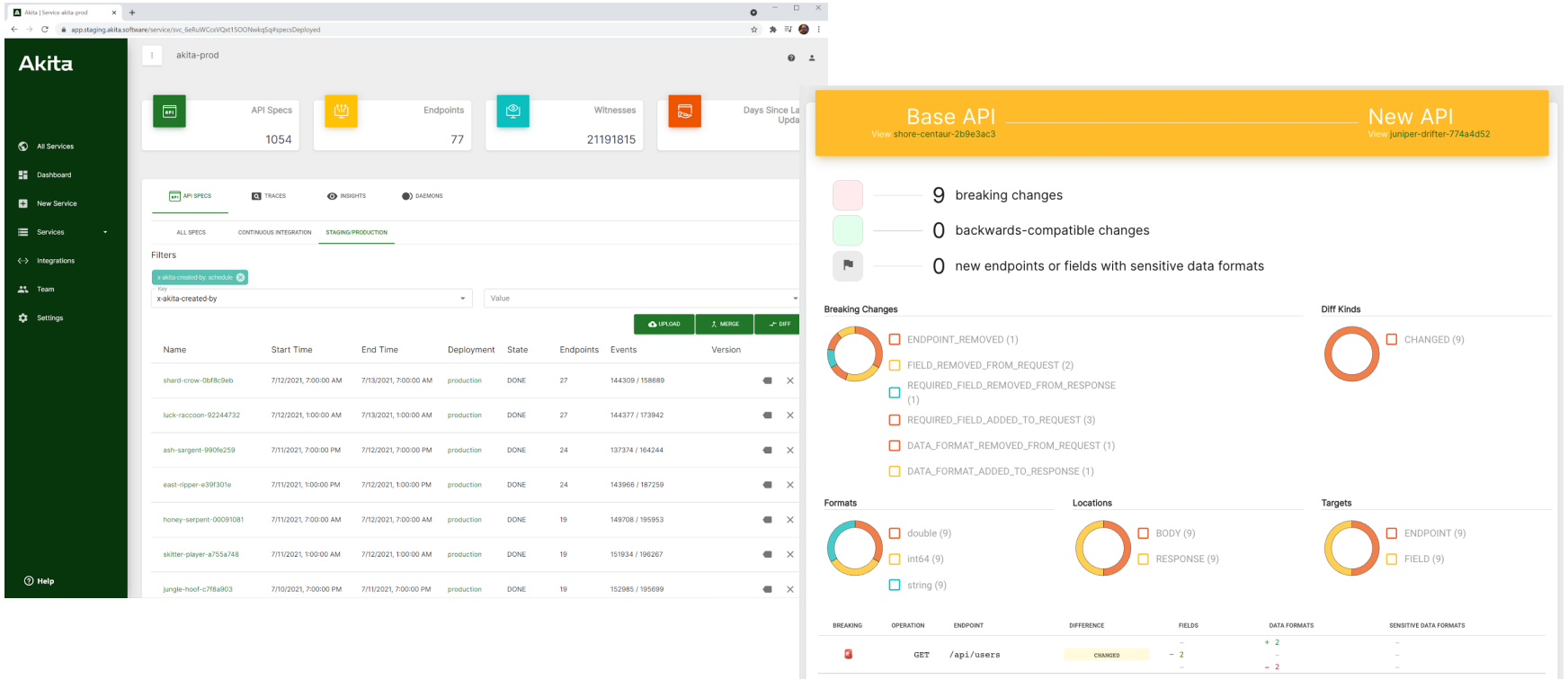


Kind of like a super smart dashboard over logging (and tracing!). Also sort of the beginning of a debugger!

Akita

Always be up-to-date on APIs. Find out when breaking changes are made to APIs.

Akita



Kind of the beginnings of something like autocomplete :D

The Future of Distributed Programming Needs Better....

Abstractions

shared nothing architecture. Of course, latency has been addressed a little bit by being able to replicate data. You can reduce that delta a little bit, but of course, not a totally solved problem. Partial failures is the thing we're going to dig into the most throughout. According to Martin Kleppmann, I think a good summary of these challenges is modern distributed systems look like this. You have a lot of different machines. They're running different processes. They only have message parsing via unreliable networks with variable delays, and the system may suffer from a host of partial failures, unreliable clocks, and process pauses.

Distributed computing is really hard to reason about. We've known this since the early '90s. We've probably always known this. This was on the radar of people building computer based systems in the '90s, a different group of people at Sun Microsystems came up with. Originally, it was seven, but number eight was added by the Java guy, James Gosling later on, but eight fallacies of distributed computing. We know today that networks are not reliable. We know today that latency is not zero. That bandwidth is not infinite, of course. That the network cannot be assumed to be secure. That topology should be assumed. We should assume that that will change.

IN SCHOOL...

in Distributed Systems

...we are led to believe that the way one must reason in a distributed system is to focus on different processes on different machines, with message-passing as the main modality for making things interact.

Sure, somebody should think at that level (e.g., if you're maintaining Zookeeper). But not everyone should be forced to reason about all of this.

Especially not when almost all programming has become distributed programming.

Data-centric abstractions

In many cases, we simply want to interact with a piece of data elsewhere. And the tools that we're given to do this is either message-passing or request-response.

This is too low-level.

Instead, why can't most end-users use data structures with built-in replication & (strong) eventual consistency?

That is, the distributed systems expert figures out how to implement some provable semantics (e.g., strong eventual consistency), and we wrap those up under a **declarative abstraction**.

Most importantly, the abstraction should be general and composable!

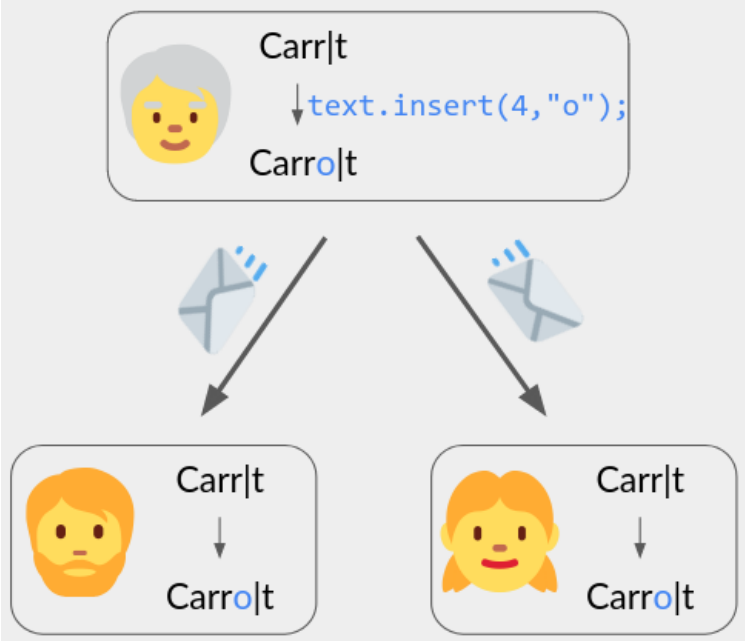
Enter: Collaborative Data Structures

Collabs are a composable abstraction backed by CRDTs at their core.

CRDTs alone cannot be combined like normal data structures.

“Eventual con-mystery”: Concurrent operations combined somehow, but might not be reasonable.

Collabs take care of this problem. You can define Collabs containing other Collabs and their operations will correctly compose in the face of concurrency and distribution.



Matthew Weidner,
CMU

@collabs/collabs

0.6.0 • Public • Published 13 days ago

Readme | Explore BETA | 3 Dependencies

Collabs

A collections library for collaborative data structures

https://www.npmjs.com/package/@collabs/collabs

Collabs is a library for building and using *collaborative data structures*. These are data structures that look like `Set`, `Map`, `Array`, etc., except they are synchronized between multiple users: when one user changes a collaborative data structure, their changes show up for every other user. You can use them to quickly build collaborative apps along the lines of Google Docs/Sheets/Slides, shared whiteboards, etc.

Quick Start

Live demos (source)

Getting Started Guide

Principles

- **Local-first:** Each user always keeps a full copy of the state on their own device and sees their own changes immediately, even when offline. They can then sync up with other users in the background. All users see the same state once they sync up, even if they made simultaneous changes (e.g., two users typing at once).
- **Network-agnostic:** `collabs` generates messages that you must eventually broadcast to all users, but how is completely up to you and your users: your own server, WebRTC, encrypted Matrix room, etc.
- **Flexible and extensible:** At its core, `collabs` is a library *for* collaborative data structures, not just a library *of* them (although we provide plenty of those too). So if our data types don't meet your needs, you can create your own or get them from third-party libraries.
- **Composable**: In particular, we provide techniques to create new types by composing existing ones. Correctness properties compose too!
- **Keep your data model and type safety:** A core feature of Collabs is that you can organize your collaborative state using reusable, strongly-typed classes. In particular, you can make a single-user app collaborative while preserving its data model and type safety, by directly replacing its frontend data types with collaborative versions.

Demos!

https://compoventuals-tests.herokuapp.com/

For example, try the collaborative whiteboard app.

What a Collabs app looks like

```
class SlideElementCRDT extends CRDTObject {
  left: LWWRegister<number>;
  top: LWWRegister<number>;
  width: LWWRegister<number>;
  height: LWWRegister<number>;
  // Rotation, borders, ...
}

class TextBoxCRDT extends SlideElementCRDT {
  text: TextCRDT;
}
// Other slide elements...

class SlideCRDT extends CRDTObject {
  elements: UniqueSetOfCRDTs<SlideElementCRDT>;
  bgColor: LWWRegister<string>;
}
```

```
class MovableListEntry<C> extends CRDTObject {
  value: C;
  positionReg: LWWRegister<ImmutablePosition>;
}

class MovableListOfCRDTs<C> extends CRDTObject
{
  state: UniqueSetOfCRDTs<MovableListEntry<C>>;
}

appState: MovableListOfCRDTs<SlideCRDT>;
```

No message passing! You program with Collabs like regular data structures!

Collabs/CRDTs aren't everything,

But they're one example of turning how you think about building a distributed application completely upside-down. We need more of these.

Looking forward from here...

The experience of developing an application that makes calls over the network needs to improve. (And it is. Just piecemeal.)

If you're responsible for an engineering org, then at first it will look like a slew of random open-source tools and startups “coming to the rescue” with a new tool or technique that they promise will help with the complexity of microservices/etc.

Key piece of advice: be open to this stuff.

I think, realistically, these disparate pieces will come together in some way. At minimum, build/configuration (pulling all of these pieces together) will become easier. Surely though, there will be effort to pull many of these goals under the umbrella of a single platform or tool.

But the main thing to remember is we *are* moving towards a better development experience, and the component pieces are being fervently developed by individuals, companies, and groups of researchers worldwide.

People/Projects to watch

Christopher Meiklejohn

CMU/DoorDash

Matthew Weidner

CMU

Geoffrey Litt

MIT

Unison

Distributed programming language

Ink & Switch

Industrial research lab

Akita

Startup

Martin Kleppmann

University of Cambridge

Peter van Hardenberg

Ink & Switch

Questions?

Thank you!
@heathercmiller