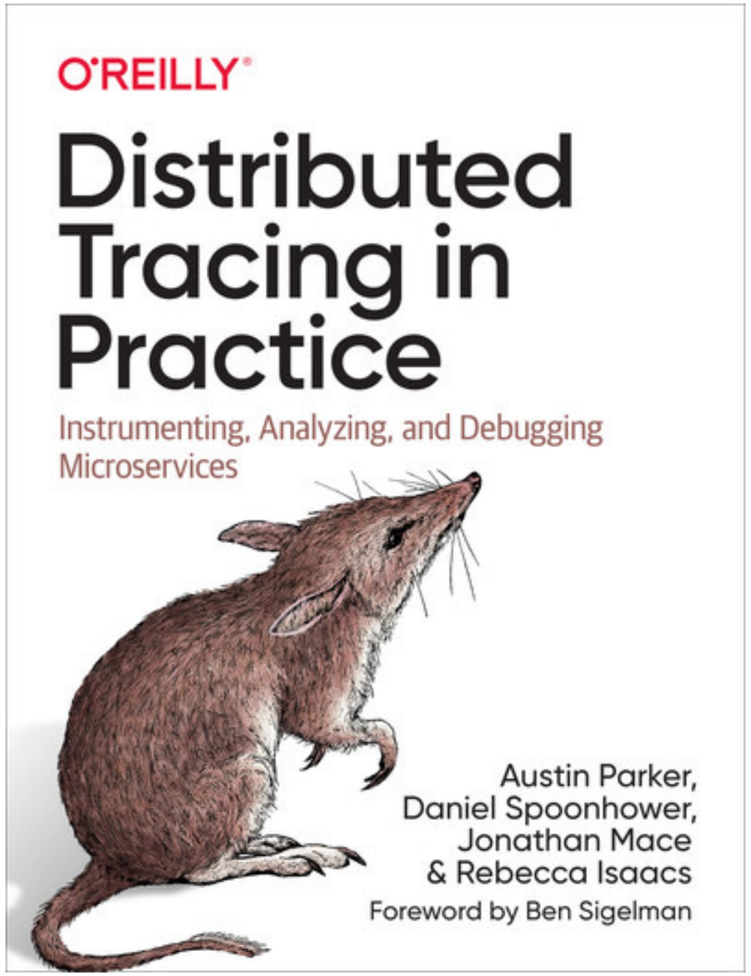
Distributed Tracing

Tejas Parikh (t.parikh@northeastern.edu)

CSYE 7125 – Advanced Cloud Computing

Northeastern University

Reference


O'REILLY
Distributed Tracing in Practice
Instrumenting, Analyzing, and Debugging Microservices
Austin Parker, Daniel Spoonhower, Jonathan Mace & Rebecca Isaacs
Foreword by Ben Sigelman

- **Publisher :** O'Reilly Media; 1st edition (May 5, 2020)
- **Language :** English
- **Paperback :** 330 pages
- **ISBN-10 :** 1492056634
- **ISBN-13 :** 978-1492056638

https://learning.oreilly.com/library/view/distributed-tracing-in/9781492056621/

What Is Distributed Tracing?

- Distributed tracing (also called distributed request tracing) is a type of correlated logging that helps you gain visibility into the operation of a distributed software system for use cases such as performance profiling, debugging in production, and root cause analysis of failures or other incidents.
- It gives you the ability to understand exactly what a particular individual service is doing as part of the whole, enabling you to ask and answer questions about the performance of your services and your distributed system as a whole.

Why Distributed Software so popular?

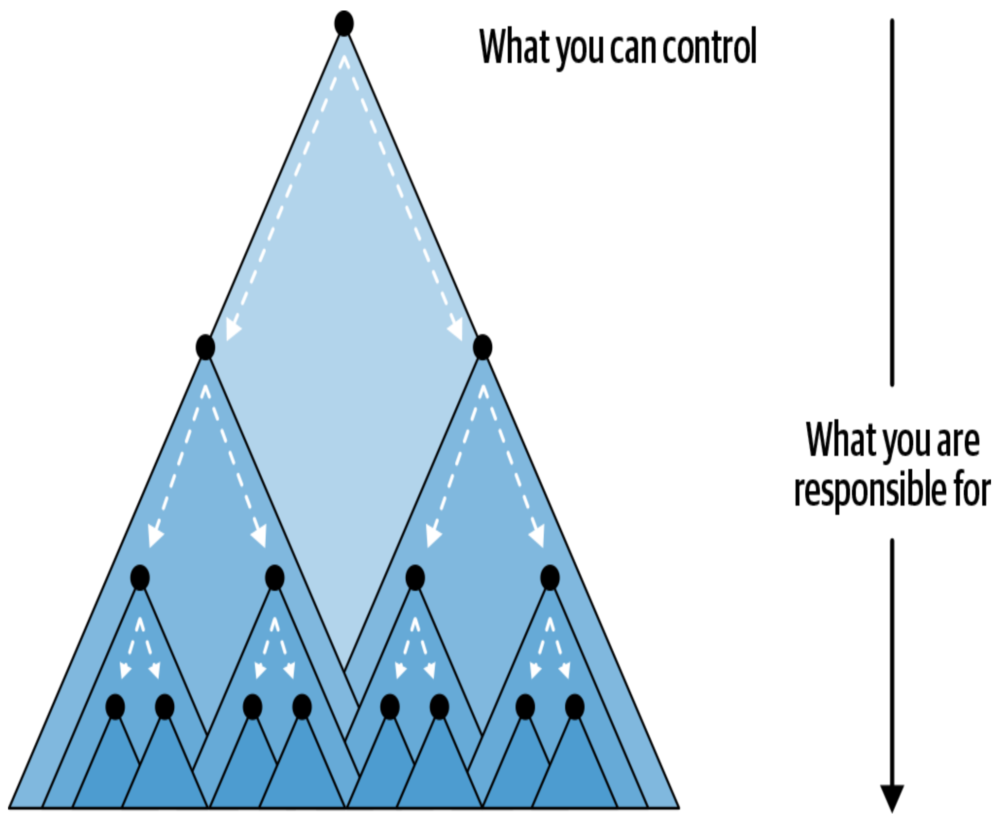
- Scalability
  - A distributed application can more easily respond to demand, and its scaling can be more efficient.
- Reliability
  - Failures in one component shouldn’t bring down the entire application.
  - Distributed applications are more resilient because they split up functions through a variety of service processes and hosts, ensuring that even if a dependent service goes offline, it shouldn’t impact the rest of the application.
- Maintainability
  - Distributed software is more easily maintainable for a couple of reasons.
  - Dividing services from each other can increase how maintainable each component is by allowing it to focus on a smaller set of responsibilities.
  - In addition, you’re freer to add features and capabilities without implementing (and maintaining) them yourself.

Deep Systems

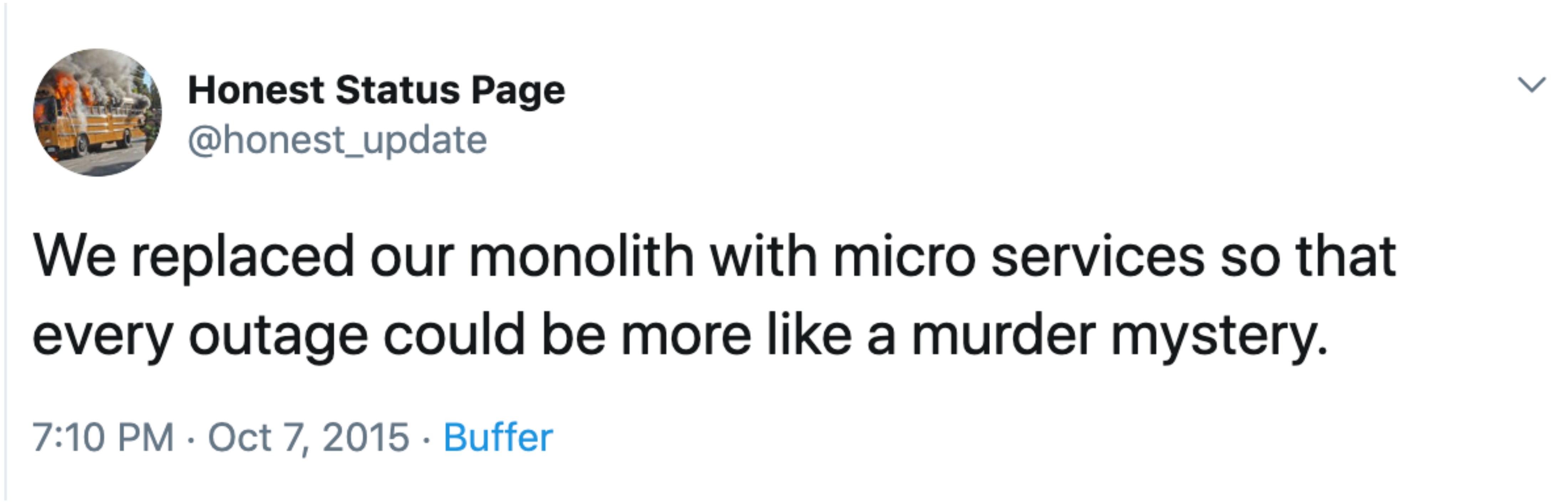
- A distributed architecture is a prime example of what software architects often call a deep system.
- These systems are notable not because of their width, but because of their complexity.
- Requests may route through three, four, fourteen, or forty different layers of services, and each of those layers may have other dependencies that you aren’t aware of.

The problem with Deep Systems

- The problem with deep systems is ultimately a human one.
- It quickly becomes unrealistic for a single human, or even a group of them, to understand enough of the services that are in the critical path of even a single request and continue maintaining it.



The Difficulties of Understanding Distributed Architectures



Honest Status Page
@honest_update

We replaced our monolith with micro services so that every outage could be more like a murder mystery.

7:10 PM · Oct 7, 2015 · Buffer

Monitoring helps determine where the body is, but it doesn't reveal why the murder occurred.

The Difficulties of Understanding Distributed Architectures

- Distributing your software presents new and exciting challenges.
- Suddenly, failures and crashes become harder to pin down.
- The service that you’re responsible for may be receiving data that’s malformed or unexpected from a source that you don’t control because that service is managed by a team halfway across the globe.
- Failures in services that you thought were rock-solid suddenly cause cascading failures and errors across all of your services.

Distributed tracing provides solutions to three major pain points

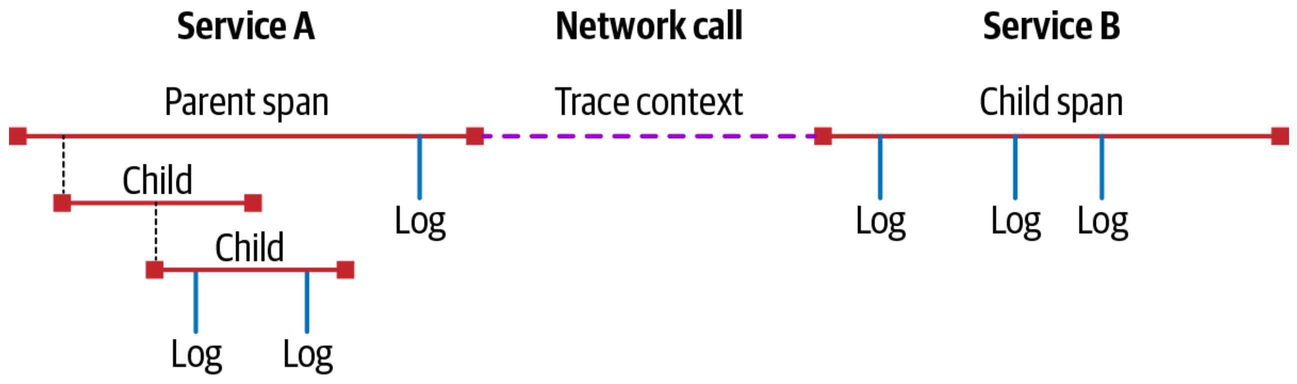
- Obfuscation
  - As your application becomes more distributed, the coherence of failures begins to decrease.
  - That is to say, the distance between cause and effect increases.
  - An outage at your cloud provider's blob storage could fan out to cause huge cascading latency for everyone, or a single difficult-to-diagnose failure at a particular service many hops away that prevents you from uncovering the proximate cause.
- Inconsistency
  - Distributed applications might be reliable overall, but the state of individual components can be much less consistent than they would be in monolithic or non-distributed applications.
  - In addition, since each component of a distributed application is designed to be highly independent, the state of those components will be inconsistent—what happens when someone does a deployment, for example? Do all of the other components understand what to do? How does that impact the overall application?
- Decentralized
  - Critical data about the performance of your services will be, by definition, decentralized.
  - How do you go looking for failures in a service when there may be a thousand copies of that service running, on hundreds of hosts?
  - How do you correlate those failures?
  - The greatest strength of distributing your application is also the greatest impediment to understanding how it actually functions!

How Does Distributed Tracing Help?

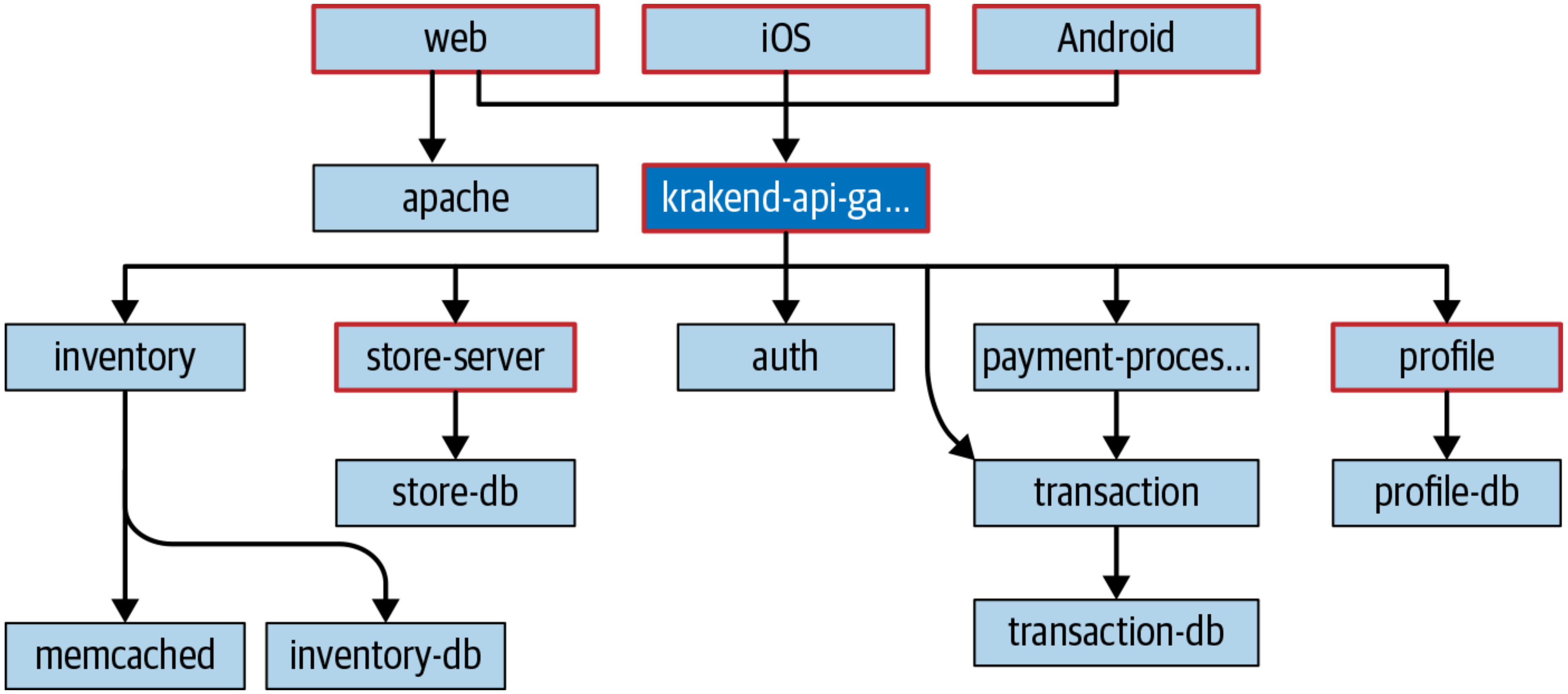
- Distributed tracing emerges as a critical tool in managing the explosion of complexity that our deep systems bring.
- It provides context that spans the life of a request and can be used to understand the interactions and shape of your architecture.
- However, these individual **traces** are just the beginning—in aggregate, traces can give you important insights about what's actually going on in your distributed system, allowing you not only to correlate interesting data about your services but also to filter and rank the importance of other types of telemetry.
- Effectively, distributed traces provide context that helps you filter problem-solving down to only things that are relevant to your investigation, so you don't have to guess and check multiple logs and dashboards.
- In this way, distributed tracing is actually at the center of a modern observability platform, and it becomes a critical component of your distributed architecture rather than an isolated tool.

What is a Trace?

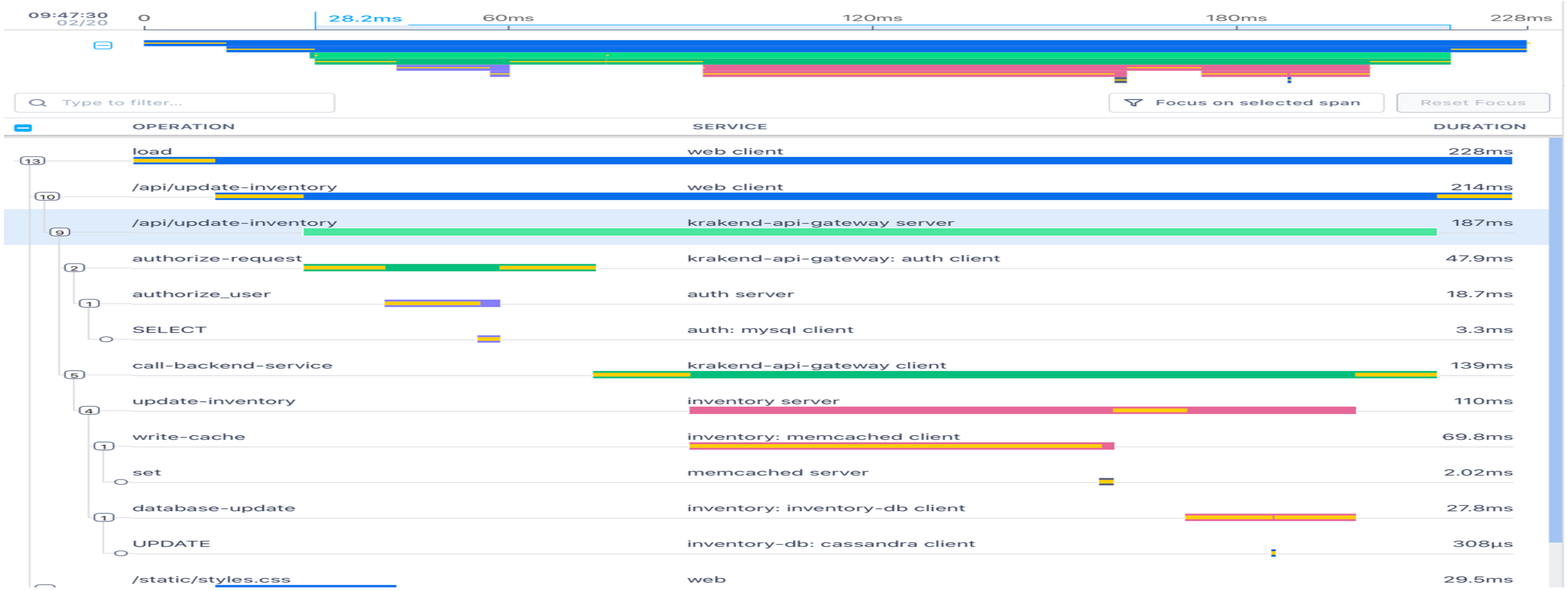
- The easiest way to understand *trace* is to think about your software in terms of requests.
- The work that each service is doing a *span*, as in the span of time that it takes for the work to occur.
- Spans can be annotated with metadata (known as attributes or tags) and events (also referred to as logs).



A service map generated from distributed trace data.



A sample trace demonstrating a request initiated by a frontend web client.



OPERATION	SERVICE	DURATION
load	web client	228ms
/api/update-inventory	web client	214ms
/api/update-inventory	krakend-api-gateway server	187ms
authorize-request	krakend-api-gateway: auth client	47.9ms
authorize_user	auth server	18.7ms
SELECT	auth: mysql client	3.3ms
call-backend-service	krakend-api-gateway client	139ms
update-inventory	inventory server	110ms
write-cache	inventory: memcached client	69.8ms
set	memcached server	2.02ms
database-update	inventory: inventory-db client	27.8ms
UPDATE	inventory-db: cassandra client	308µs
/static/styles.css	web	29.5ms

The Benefits of Tracing

- Distributed tracing can transform the way that you develop and deliver software, no doubt about it. It has benefits not only for software quality, but for your organization’s health.
- Distributed tracing can improve developer productivity and your development output. It is the best and easiest way for developers to understand the behavior of distributed systems in production. You will spend less time troubleshooting and debugging a distributed system by using distributed tracing than you would without it, and you’ll discover problems you wouldn’t otherwise realize you had.
- Distributed tracing supports modern polyglot development. Since distributed tracing is agnostic to your programming language, monitoring vendor, and runtime environment, you can propagate a single trace from an iOS-native client through a C++ high-performance proxy through a Java or C# backend to a web-scale database and back, all visualized in a single place, using a single tool. No other set of tools allows you this freedom and flexibility.
- Distributed tracing reduces the overhead required for deployments and rollbacks by quickly giving you visibility into changes. This not only reduces the mean time to resolution of incidents, but decreases the time to market for new features and the mean time to detection of performance regressions. This also improves communication and collaboration across teams because your developers aren’t siloed into a particular monitoring stack for their slice of the pie—everyone, from frontend developers to database nerds, can look at the same data to understand how changes impact the overall system.

Additional Resources

See Lecture Page