MICROSERVICES / MONITORING

Debugging Microservices: Lessons from Google, Facebook, Lyft

3 Jul 2018 9:43am, by Joab Jackson

It was about five years ago when Lyft engineers started seeing operational problems in its move to microservices. Services were getting increasingly difficult to troubleshoot and, as a result, developers were starting not to trust microservices in production at all.

Lyft's Matt Klein.

The chief problem was the lack of observability, noted Matt Klein, Lyft software engineer who created the open source Envoy service mesh, primarily to deal with this issue.

"If the network fails and you can't understand where it is failing, it is very difficult to do anything about it," Klein said, before an audience at the QCon developer conference last week in New York.

The company was then moving to a microservices architecture, one that would allow developers to write their components in their favored frameworks and languages, be it PHP, JavaScript, Python or Go.

But when things went awry, it was hard to figure why. Was it the Internet? The load balancer? The back-end database. Sure, a dev could scroll through the error logs, but there were so many, all in different formats, and they all held so much information. And how could you trace a problem as it bounced from service to service?

"From the operational standpoint trying to stitch all this together is very difficult," Klein said.

Envoy came about as a way to make their job easier, without them actually worry about network, authentication drivers or other software to deal with the network layer. "Every minute application developers are not writing business logic is time wasted," he said. With Envoy, "we like to abstract as much as possible, but when it can't be abstracted, because things fail, we like to help people figure out what the source of the failure is, so we can actually fix it."

A service mesh, Envoy co-locates a sidecar to every service. When one service wants to talks with another, it is the sidecar that makes this connection, with the other service's sidecar. This way, they can share traces, logs, and stats. Each service only knows about its own sidecar and leaves Envoy itself to worry about factors such as service discovery, load balancing, rate limiting, circuit breaking, retries, and other network-centric and scalability tasks.

"All those things have been hidden from the application developer, which is a very powerful paradigm," Klein said.

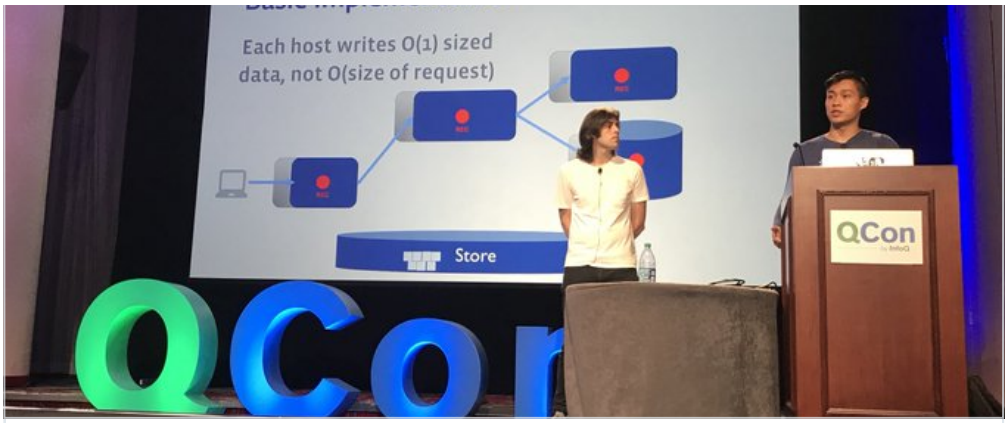
Follow the Request Trace

"Distributed tracing is the best tool in the engineer's toolbox for tackling elusive hard-to-pint problems in a complex architecture, said Facebook engineer Haozhe Gao in another talk at QCon, about Facebook's internal distributed tracing tool, Canopy.

Just like Lyft, Facebook knows microservices. Simply loading the home page will call thousands of remote procedure calls (RPCs). How do you debug a problem, for instance, that only happens to 10 percent of the users? "Investigating edge behavior across multiple hosts gets very challenging," he said.

One common technique Facebook uses is measuring latency at RPC boundaries, often by sampling, or by aggregation. Though useful, RPC samples of endpoints own they only gives you part of the picture, however. Enter distributed tracing, which gathers a small amount of performance information from each service involved in a request, and then stores this information as a "trace," and typically represented as an RPC graph, Gao explained.

A typical distributed service will have hooks on each service it measures. Each hook can initiate a trace or discontinue one. A hook that initiates a service must signal to the backend service that a trace has been started. This usually done attaching a header to the RPC call, or a small metadata field onto the request. "Whatever it is you choose, choose something in the transport layer because application layer logic gets messy," Gao said.



The New Stack
@thenewstack

To monitor its thousands of services, Facebook captures about a billion traces a day (about ~100TB collected), a dynamic sampling of the total number of interactions per day — @Facebook's Haozhe Gao and Joe O'Neill #QConNYC

2 5:35 PM - Jun 27, 2018

See The New Stack's other Tweets

Each service in the chain then gets the message to turn tracing on. The performance data gathered for the trace itself is not passed from service to service, but rather uploaded to a data store directly. "We'd rather have each service collect its own performance information," he said, noting passing this info from service to service would bloat the message size.

With this complete RPC call-tree and associated latency info in hand, "You can tell exactly how time is distributed in subgroups for that group of requests," he said. "You're spending less time correlating logs and piecing stuff together, and more time looking at a holistic picture of performance.

Currently, Facebook collects about a billion traces — about 100TB of data — per day, noted Facebook Engineer Joe O'Neill, who joined Gao in the presentation. This is just a small percentage of the traces that could be done, but for concerns of bandwidth and storage, a complete record is not necessary.

"For us, relying on statistically significant sample size works out pretty well. Most APIs have similar enough request loads, and when you have outliers, you can actually capture those within your sample size," O'Neill said.

specific level of needed granularity.

No Blind Clicking

As your system grows more complex, and your knowledge of what can go wrong increases, you may be tempted to expand a dashboard with more metrics representing outages. This is a bad idea, advised Google Site Reliability Engineer (SRE) Liz Fong-Jones. Too many dashboards leads to cognitive overload, and as the SRE just blindly looks through a set of a set of visualized queries, looking for patterns. It's wasted time, she warned.

"No two outages are identical, so it is important to figure out the common patterns that you can use to generically debug any outage, and what are the things specific to that one outage," said Fong-Jones, during a QCon talk with fellow SRE Adam Mckaig that offered some tips in better framing the debugging process.

Their talk was more about techniques than tools. Google itself still uses internally-built monitoring tools, notably Panopticon (nicknamed "Pcon" internally) and Monarch. Some of these capabilities are available, or will be available soon, in its customer-facing metrics service Stackdriver.

> "No two outages are identical, so it is important to figure out the common patterns that you can use to generically debug any outage, and what are the things specific to that one outage," — Liz Fong-Jones

In its rapid growth, Google created for itself an "explosion of metrics" Fong-Jones said. It was generating and collecting metrics coming from applications running across thousands of servers. Google defines success, and failure to claim success, through pre-defined service level objectives (SLOs), which measures is users are able to use the service.

At Google, an alert is to sent to an SRE based on failure to meet an SLO metric. Then the mitigation process begins, After doing any quick fixes to mitigate the issue, the SRE should then form a hypothesis as to what may be going wrong. In complex systems, the first hypothesis rarely the correct one, so most SREs must formulate multiple hypothesizes, and, ideally, as quickly as possible.

everything that is working correctly — replicas in other availability zones, working replicas in the same zone etc. This approach can catch obvious problems like bad nodes.

- **Dynamic Data Joins** bring in external metrics from elsewhere in the system This approach helps when "there is no path between the alert and the actual problem," Mckaig said, citing an example of a specific version of a kernel being run by some hosts that introduced latency. The standard service metrics don't typically capture kernel versions of the host, but another table can be joined in by the SRE to see the correlation.
- **Exemplars** is a Google technique that uses selective full-trace sampling to get at the problem area faster than bisection. When the data is visualized on a heat map, buckets of metrics around a specific operation can be quickly identified in the problem areas, allowing the SRE to pinpoint the bad task. This approach is used by Honeycomb, Circonis and Grafana user interfaces (and soon Stackdriver).

Google has a set of standing queries (or "pre-computations" in Googlespeak), of aggregate ops that compiled throughout the day, making it easy to filter extraneous data. In troubleshooting, an SRE can pull in one of these queries for a quick view of what may going wrong, then call up that query itself to modify it to just the desired attributes. This query will be slower than the pre-run canned ones because it is being executed in real-time, but the results will be more specific, Mckaig notes.

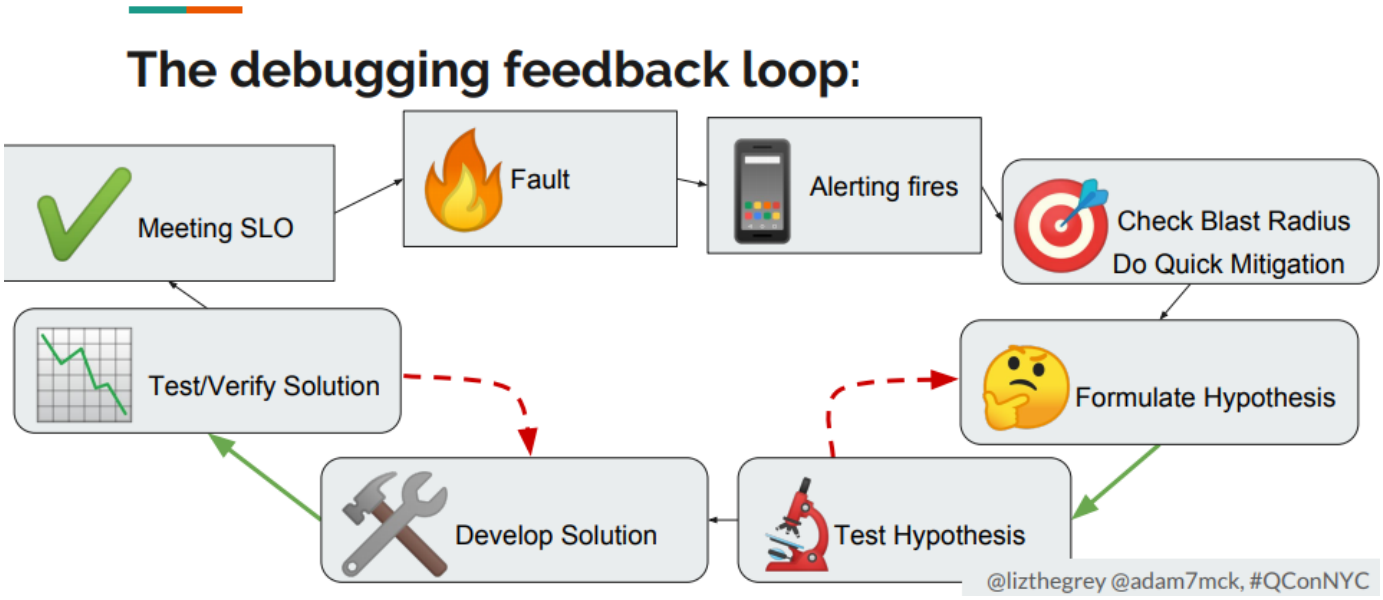
When you find a new type of performance issue, the temptation is to add a new set of metrics to a dashboard. Most of the time this is not a good idea. Overly busy dashboards can quickly lead to cognitive overload — Google's @lizthegrey on #microservices debugging #qconnyc

66 7:29 PM - Jun 28, 2018 · New York Marriott Marquis

17 people are talking about this

Fong-Jones suggested that organizations should build up a library of common metrics queries that their own SREs can easily tap into. This will allow engineers to more quickly find trouble areas, without sending out blind queries.

For tracing, Google collects operational data in high fidelity, then uses an algorithm to determine which data to keep and which to delete. The algo filtering out duplicate traces, or duplicate replica performances, through weighted averages. For instance, YouTube uses exemplars to catch performance video encoding performance, capturing a specific task ID from each latency bucket and time window.



QCon will post videos of these talks online within the next few months.

Feature image: Software engineers Haozhe Gao (Left) and Joe O'Neill, QCon New York, 2018.

ANALYSIS