Chapter 4. The Three Pillars of Observability

Logs, metrics, and traces are often known as the three pillars of observability. While plainly having access to logs, metrics, and traces doesn't necessarily make systems more observable, these are powerful tools that, if understood well, can unlock the ability to build better systems.

Event Logs

An *event log* is an immutable, timestamped record of discrete events that happened over time. Event logs in general come in three forms but are fundamentally the same: a timestamp and a payload of some context. The three forms are:

Plaintext

A log record might be free-form text. This is also the most common format of logs.

Structured

Much evangelized and advocated for in recent days. Typically, these logs are emitted in the JSON format.

Binary

Think logs in the Protobuf format, MySQL binlogs used for replication and point-in-time recovery, systemd journal logs, the `pflog` format used by the BSD firewall `pf` that often serves as a frontend to `tcpdump`.

Debugging rare or infrequent pathologies of systems often entails debugging at a very fine level of granularity. Event logs, in particular, shine when it comes to providing valuable insight along with ample context into the long tail that averages and percentiles don't surface. As such, event logs are especially helpful for uncovering emergent and unpredictable behaviors exhibited by components of a distributed system.

Failures in complex distributed systems rarely arise because of one specific event happening in one specific component of the system. Often, various possible triggers across a highly interconnected graph of components are involved. By simply looking at discrete events that occurred in any given system at some point in time, it becomes impossible to determine all such triggers. To nail down the different triggers, one needs to be able to do the following:

- Start with a symptom pinpointed by a high-level metric or a log event in a specific system
- Infer the request lifecycle across different components of the distributed architecture
- Iteratively ask questions about interactions among various parts of the system

In addition to inferring the fate of a request throughout its lifecycle (which is usually short lived), it also becomes necessary to be able to infer the fate of a system as a whole (measured over a duration that is orders of magnitudes longer than the lifecycle of a single request).

Traces and metrics are an abstraction built on top of logs that pre-process and encode information along two orthogonal axes, one being request-centric (trace), the other being system-centric (metric).

The Pros and Cons of Logs

Logs are, by far, the easiest to generate. The fact that a log is just a string or a blob of JSON or typed key-value pairs makes it easy to represent any data in the form of a log line. Most languages, application frameworks, and libraries come with support for logging. Logs are also easy to instrument, since adding a log line is as trivial as adding a print statement. Logs perform really well in terms of surfacing highly granular information pregnant with rich local context, so long as the search space is localized to events that occurred in a single service.

The utility of logs, unfortunately, ends right there. While log *generation* might be easy, the performance idiosyncrasies of various popular logging libraries leave a lot to be desired. Most performant logging libraries allocate very little, if any, and are extremely fast. However, the default logging libraries of many languages and frameworks are not

the cream of the crop, which means the application as a whole becomes susceptible to suboptimal performance due to the overhead of logging. Additionally, log messages can also be lost unless one uses a protocol like RELP to guarantee reliable delivery of messages. This becomes especially important when log data is used for billing or payment purposes.

RELP ISN'T A SILVER BULLET

RELP is a protocol that uses a command-response model (the command and the response is called a RELP *transaction*). The RELP client issues commands, and the RELP server responds to these commands.

The RELP server is designed to throttle the number of outstanding commands to conserve resources. Opting to use RELP means making the choice to apply back-pressure and block the producers if the server can't process the commands being issued fast enough.

While such stringent requirements might apply to scenarios when every log line is critical or is legally required for auditing purposes, monitoring and debugging rarely, if ever, calls for such strict guarantees and the attendant complexity.

Last, unless the logging library can dynamically sample logs, logging excessively has the capability to adversely affect application performance as a whole. This is exacerbated when the logging isn't asynchronous and request processing is blocked while writing a log line to disk or `stdout`.

TO SAMPLE, OR NOT TO SAMPLE?

An antidote often proposed to the cost overhead of logging is to sample intelligently. *Sampling* is the technique of picking a small subset of the total population of event logs generated to be processed and stored. This subset is expected to be a microcosm of the corpus of events generated in a system.

Sampling isn't without its fair share of issues. For one, the efficacy of the sampled dataset is contingent on the chosen keys or features of the dataset based on which the sampling decision is made. Furthermore, for most online services, it becomes necessary to determine how to dynamically sample so that the sample rate is self-adjusting based on the shape of the incoming traffic. Many latency-sensitive systems have stringent bounds on, for instance, the amount of CPU time that can be spent on emitting observability data. In such scenarios, sampling can prove to be computationally expensive.

No talk on sampling is complete without mentioning probabilistic data structures capable of storing a summary of the entire dataset. In-depth discussion of these techniques is outside the scope of this report, but there are good O'Reilly resources for those curious to learn more.

On the processing side, raw logs are almost always normalized, filtered, and processed by a tool like Logstash, fluentd, Scribe, or Heka before they're persisted in a data store like Elasticsearch or BigQuery. If an application generates a large volume of logs, then the logs might require further buffering in a broker like Kafka before they can be processed by Logstash. Hosted solutions like BigQuery have quotas one cannot exceed.

On the storage side, while Elasticsearch might be a fantastic search engine, running it carries a real operational cost. Even if an organization is staffed with a team of operations engineers who are experts in operating Elasticsearch, other drawbacks may exist. Case in point: it's not uncommon to see a sharp downward slope in the graphs in Kibana, not because traffic to the service is dropping, but because Elasticsearch cannot keep up with the indexing of the sheer volume of data being thrown at it. Even if log ingestion processing isn't an issue with Elasticsearch, no one I know

of seems to have fully figured out how to use Kibana's UI, let alone *enjoy* using it.

Logging as a Stream Processing Problem

Event data isn't used exclusively for application performance and debugging use cases. It also forms the source of all analytics data. This data is often of tremendous utility from a business intelligence perspective, and usually businesses are willing to pay for both the technology and the personnel required to make sense of this data in order to make better product decisions.

The interesting aspect here is that there are striking similarities between questions a business might want answered and questions software engineers and SREs might want answered during debugging. For example, here is a question that might be of business importance:

> *Filter to outlier countries from where users viewed this article fewer than 100 times in total.*

Whereas, from a debugging perspective, the question might look more like this:

> *Filter to outlier page loads that performed more than 100 database queries.*
>
> *Show me only page loads from France that took more than 10 seconds to load.*

Both these queries are made possible by events. Events are structured (optionally typed) key-value pairs. Marrying business information along with information about the lifetime of the request (timers, durations, and so forth) makes it possible to repurpose analytics tooling for observability purposes.

Log processing neatly fits into the bill of Online Analytics Processing (OLAP). Information derived from OLAP systems is not very different from information derived for debugging or performance analysis or anomaly detection at the edge of the system. One way to circumvent the issue with ingest delay in Elasticsearch—or indexing-based stores in gen-

eral—is by treating log processing as a stream processing problem to deal with large data volumes by using minimal indexing.

Most analytics pipelines use Kafka as an event bus. Sending enriched event data to Kafka allows one to search in real time over streams with KSQL, a streaming SQL engine for Kafka.

Enriching business events that go into Kafka with additional timing and other metadata required for observability use cases can be helpful when repurposing existing stream processing infrastructures. A further benefit this pattern provides is that this data can be expired from the Kafka log regularly. Most event data required for debugging purposes are valuable only for a relatively short period of time after the event has been generated, unlike any business-centric information that is evaluated and persisted by an ETL job. Of course, this makes sense only when Kafka already is an integral part of an organization. Introducing Kafka into a stack purely for real-time log analytics is a bit of an overkill, especially in non-JVM shops without any significant JVM operational expertise.

An alternative is Humio, a hosted and on-premises solution that treats log processing as a stream processing problem. Log data can be streamed from each machine directly into Humio without any pre-aggregation. Humio uses sophisticated compression algorithms to effectively compress and retrieve the log data. Instead of *a priori* indexing, Humio allows for real-time, complex queries on event stream data. Since Humio supports text-based logs (the format that the vast majority of developers are used to grepping), ad hoc schema on reads allows users to iteratively and interactively query log data. Yet another alternative is Honeycomb, a hosted solution based on Facebook's Scuba that takes an opinionated view of accepting only structured events, but allows for read-time aggregation and blazing fast real-time queries over millions of events.

Metrics

Metrics are a numeric representation of data measured over intervals of time. Metrics can harness the power of mathematical modeling and prediction to derive knowledge of the behavior of a system over intervals of time in the present and future.

Since numbers are optimized for storage, processing, compression, and retrieval, metrics enable longer retention of data as well as easier querying. This makes metrics perfectly suited to building dashboards that reflect historical trends. Metrics also allow for gradual reduction of data resolution. After a certain period of time, data can be aggregated into daily or weekly frequency.

The Anatomy of a Modern Metric

One of the biggest drawbacks of historical time-series databases has been the *identification* of metrics that didn't lend itself very well to exploratory analysis or filtering.

The hierarchical metric model and the lack of tags or labels in older versions of Graphite especially hurt. Modern monitoring systems like Prometheus and newer versions of Graphite represent every time series using a metric name as well as additional key-value pairs called *labels*. This allows for a high degree of dimensionality in the data model.

A metric in Prometheus, as shown in Figure 4-1, is identified using both the metric name and the labels. The actual data stored in the time series is called a *sample*, and it consists of two components: a *float64* value and a millisecond precision timestamp.

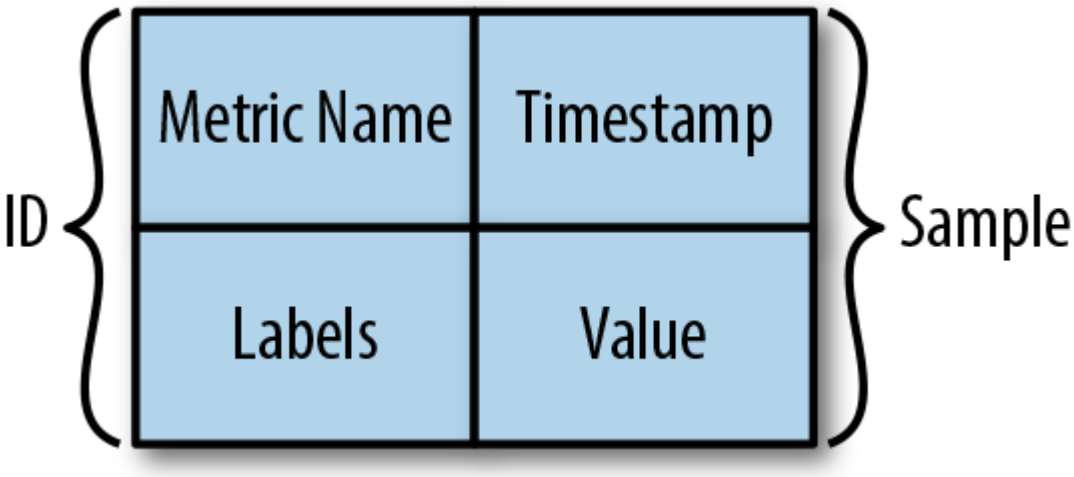

Figure 4-1. A Prometheus metric sample

It's important to bear in mind that metrics in Prometheus are immutable. Changing the name of the metric or adding or removing a label will result in a new time series.

Advantages of Metrics over Event Logs

By and large, the biggest advantage of metrics-based monitoring over logs is that unlike log generation and storage, metrics transfer and storage has a constant overhead. Unlike logs, the cost of metrics doesn't increase in lockstep with user traffic or any other system activity that could result in a sharp uptick in data.

With metrics, an increase in traffic to an application will not incur a significant increase in disk utilization, processing complexity, speed of visualization, and operational costs the way logs do. Metrics storage increases with more permutations of label values (e.g., when more hosts or containers are spun up, or when new services get added or when existing services get instrumented more), but client-side aggregation can ensure that metric traffic doesn't increase proportionally with user traffic.

NOTE

Client libraries of systems like Prometheus aggregate time-series samples in-process and submit them to the Prometheus server upon a successful scrape (which by default happens once every few seconds and can be configured). This is unlike statsd clients that send a UDP packet every time a metric is recorded to the statsd daemon (resulting in a directly proportional increase in the number of metrics being submitted to statsd compared to the traffic being reported on!).

Metrics, once collected, are more malleable to mathematical, probabilistic, and statistical transformations such as sampling, aggregation, summarization, and correlation. These characteristics make metrics better suited to report the overall health of a system.

Metrics are also better suited to trigger alerts, since running queries against an in-memory, time-series database is far more efficient, not to mention more reliable, than running a query against a distributed system like Elasticsearch and then aggregating the results before deciding if an alert needs to be triggered. Of course, systems that strictly query only in-memory structured event data for alerting might be a little less expensive than Elasticsearch. The downside here is that the operational overhead of running a large, clustered, in-memory database, even if it were open source, isn't something worth the operational trouble for most organiza-

tions, especially when there are far easier ways to derive equally actionable alerts. Metrics are best suited to furnish this information.

The Drawbacks of Metrics

The biggest drawback with both application logs and application metrics is that they are *system* scoped, making it hard to understand anything else other than what's happening inside a particular system. Sure, metrics can also be request scoped, but that entails a concomitant increase in label fan-out, which results in an increase in metric storage.

With logs without fancy joins, a single line doesn't give much information about what happened to a request across all components of a system. While it's possible to construct a system that correlates metrics and logs across the address space or RPC boundaries, such systems require a metric to carry a UID as a label.

Using high cardinality values like UIDs as metric labels can overwhelm time-series databases. Although the new Prometheus storage engine has been optimized to handle time-series churn, longer time-range queries will still be slow. Prometheus was just an example. All popular existing time-series database solutions suffer performance under high cardinality labeling.

When used optimally, logs and metrics give us complete omniscience into a silo, but nothing more. While these might be sufficient for understanding the performance and behavior of individual systems, both stateful and stateless,they aren't sufficient to understand the lifetime of a request that traverses multiple systems.

Distributed tracing is a technique that addresses the problem of bringing visibility into the lifetime of a request across several systems.

Tracing

A *trace* is a representation of a series of causally related distributed events that encode the end-to-end request flow through a distributed system.

Traces are a representation of logs; the data structure of traces looks almost like that of an event log. A single trace can provide visibility into both the path traversed by a request as well as the structure of a request. The path of a request allows software engineers and SREs to understand the different services involved in the path of a request, and the structure of a request helps one understand the junctures and effects of asynchrony in the execution of a request.

Although discussions about tracing tend to pivot around its utility in a microservices environment, it's fair to suggest that any sufficiently complex application that interacts with—or rather, contends for—resources such as the network, disk, or a mutex in a nontrivial manner can benefit from the advantages tracing provides.

The basic idea behind tracing is straightforward—identify specific points (function calls or RPC boundaries or segments of concurrency such as threads, continuations, or queues) in an application, proxy, framework, library, runtime, middleware, and anything else in the path of a request that represents the following:

- Forks in execution flow (OS thread or a green thread)
- A hop or a fan out across network or process boundaries

Traces are used to identify the amount of work done at each layer while preserving causality by using *happens-before* semantics. Figure 4-2 shows the flow of a single request through a distributed system. The trace representation of this request flow is shown in Figure 4-3. A trace is a directed acyclic graph (DAG) of *spans*, where the edges between spans are called *references*.

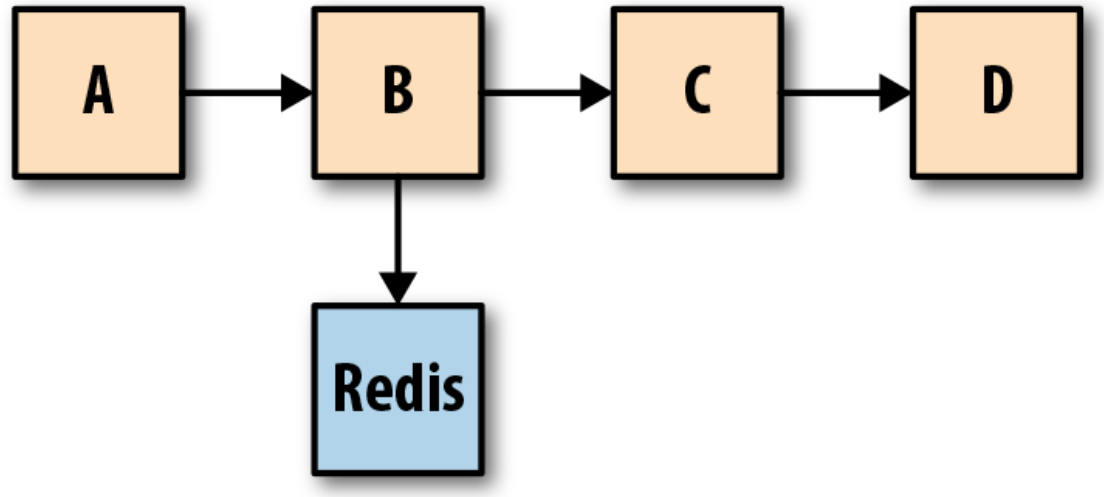

Figure 4-2. A sample request flow diagram

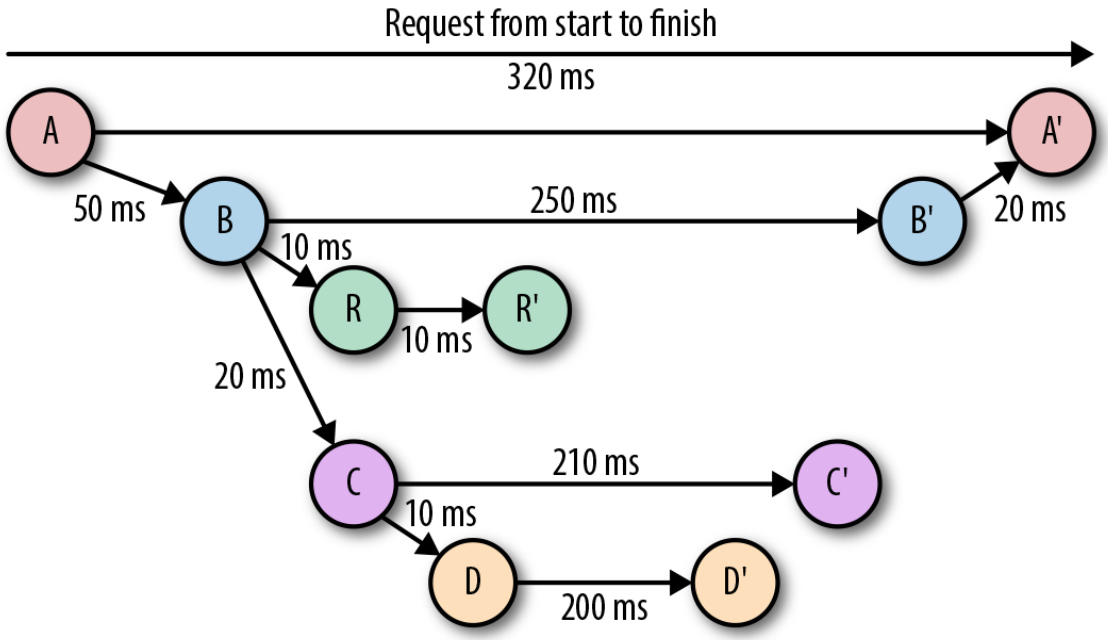

Figure 4-3. The various components of a distributed system touched during the lifecycle of a request, represented as a directed acyclic graph

When a request begins, it's assigned a globally unique ID, which is then propagated throughout the request path so that each point of instrumentation is able to insert or enrich metadata before passing the ID around to the next hop in the meandering flow of a request. Each hop along the flow is represented as a span (Figure 4-4). When the execution flow reaches the instrumented point at one of these services, a record is emitted along with metadata. These records are usually asynchronously logged to disk before being submitted out of band to a collector, which then can reconstruct the flow of execution based on different records emitted by different parts of the system.

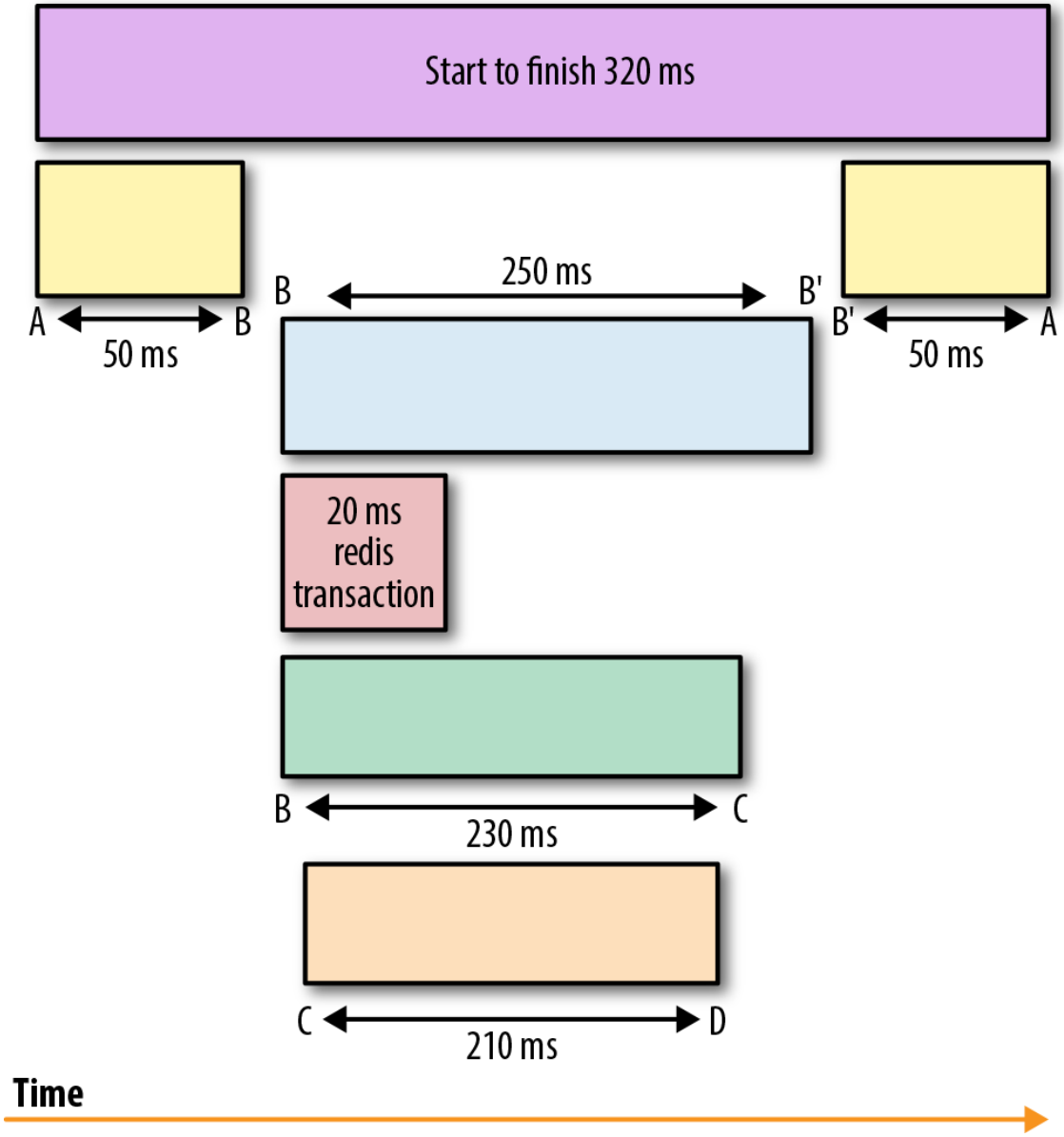

Figure 4-4. A trace represented as spans: span A is the root span, span B is a child of span A

Collecting this information and reconstructing the flow of execution while preserving causality for retrospective analysis and troubleshooting enables one to better understand the lifecycle of a request.

Most importantly, having an understanding of the entire request lifecycle makes it possible to debug requests spanning multiple services to pinpoint the source of increased latency or resource utilization. For example, Figure 4-4 indicates that the interaction between service C and service D was what took the longest. Traces, as such, largely help one understand the *which* and sometimes even the *why* (e.g., *which* component of a system is even touched during the lifecycle of a request and is slowing the response?).

The use cases of distributed tracing are myriad. While used primarily for inter service dependency analysis, distributed profiling, and debugging

steady-state problems, tracing can also help with chargeback and capacity planning.

Zipkin and Jaeger are two of the most popular OpenTracing-compliant open source distributed tracing solutions. (OpenTracing is a vendor-neutral spec and instrumentation libraries for distributed tracing APIs.)

The Challenges of Tracing

Tracing is, by far, the hardest to retrofit into an existing infrastructure, because for tracing to be truly effective, every component in the path of a request needs to be modified to propagate tracing information. Depending on whom you ask, you'd either be told that having gaps in the flow of a request doesn't outweigh the cons (since adding tracing piecemeal is seen as better than having no tracing at all, as having partial tracing helps eke out nuggets of knowledge from the fog of war) or be told that these gaps are blind spots that make debugging harder.

The second problem with tracing instrumentation is that it's not sufficient for developers to instrument their code alone. A large number of applications in the wild are built using open source frameworks or libraries that might require additional instrumentation. This becomes all the more challenging at places with polyglot architectures, since every language, framework, and wire protocol with widely disparate concurrency patterns and guarantees needs to cooperate. Indeed, tracing is most successfully deployed in organizations that use a core set of languages and frameworks uniformly across the company.

The cost of tracing isn't quite as catastrophic as that of logging, mainly because traces are almost always sampled heavily to reduce runtime overhead as well as storage costs. Sampling decisions can be made:

- At the start of a request before any traces are generated
- At the end, after all participating systems have recorded the traces for the entire course of the request execution
- Midway through the request flow, when only downstream services would then report the trace

All approaches have their own pros and cons, and one might even want to use them all.

Service Meshes: A New Hope for the Future?

While tracing has been difficult to implement, the rise of service meshes make integrating tracing functionality almost effortless. Data planes of service meshes implement tracing and stats collections at the proxy level, which allows one to treat individual services as blackboxes but still get uniform and thorough observability into the mesh as a whole. Applications that are a part of the mesh will still need to forward headers to the next hop in the mesh, but no additional instrumentation is necessary.

Lyft famously got tracing support for every last one of its services by adopting the service mesh pattern, and the only change required at the application layer was to forward certain headers. This pattern is incredibly useful for retrofitting tracing into existing infrastructures with the least amount of code change.

Conclusion

Logs, metrics, and traces serve their own unique purpose and are complementary. In unison, they provide maximum visibility into the behavior of distributed systems. For example, it makes sense to have the following:

- A counter and log at every major entry and exit point of a request
- A log and trace at every decision point of a request

It also makes sense to have all three semantically linked such that it becomes possible at the time of debugging:

- To reconstruct the codepath taken by reading a trace
- To dervive request or error ratios from any single point in the codepath

Sampling exemplars of traces or events and correlating to metrics unlocks the ability to click through a metric, see examples of traces, and inspect the request flow through various systems. Such insights gleaned

from a combination of different observability signals becomes a must-have to truly be able to debug distributed systems.