DOI:10.1145/3378931

To view the accompanying paper, visit doi.acm.org/10.1145/3378933

Technical Perspective

A Perspective on Pivot Tracing

By Rebecca Isaacs

DISTRIBUTED SYSTEMS ARE difficult to manage at the best of times: diagnosis, debugging, capacity planning, and configuration of runtime properties like timeouts or service-level objective (SLO) thresholds, are made more challenging by the extra complexity that arises from distribution. Throw into the mix that a single request will be serviced by multiple independent microservices, and these challenges compose and multiply.

Yet this type of serving environment is completely normal—just about every online service uses a collection of distinct, communicating functions to fulfil each user request. For example, the sale of a single item on a shopping site might involve an authentication service, a bot detection service, an inventory management service, and a payments service, each of which will be sharded *N* ways and likely using a caching layer in front of (distributed) persistent storage. With distribution comes scale: requests consisting of hundreds, or even thousands, of nested RPCs are not unusual in Web services. Standard mechanisms for batching, pipelining, concurrency, fault tolerance, hedging of requests, load balancing, and other such in-band, dynamic, control systems further exacerbate the difficulties of understanding system behavior.

In such an environment, how do service providers debug their systems? If the shopping cart checkout request latency exceeds the 99th percentile, how can we identify which microservice was responsible and why? One important tool for tackling this kind of problem, on par with, and complementary to logs, counters, and metrics, is tracing.

A typical end-to-end request tracing system relies on the RPC subsystem to propagate a unique request identifier between microservices, and thus tie together causally related service invocations. A trace will also capture metadata about the request, collected at each hop along the way—details like the URI string or a client identifier, the name and IP address of each host, and often performance metrics such as how much CPU the request consumed at each component.

This design is simple but has an inherent tension between generality and cost. Most tracing systems pick a point in this trade-off space in which every service instance generates records locally and transmits them directly to a collector that groups records from across the system by trace identifier. With this approach, the set of queries that can be run (offline) against the traces is unconstrained, but the system can produce vast amounts of data with correspondingly high cost. As a result, requests are traced at a low sampling rate (.01% or lower is typical in production), which means rare events, often critical for detection and diagnosis of problems, may be missed.

Pivot Tracing, the system described in the following paper, chooses a different trade-off. Instead of eagerly handing trace records off to a collector for long-term storage and future processing, it installs continuous queries, on demand, inside the distributed system itself, and dynamically enables instrumentation at *tracepoints* to record exactly the information needed to answer the currently active queries. By this design, Pivot Tracing favors specificity in return for low cost, while removing the need to down-sample requests. A particularly appealing aspect is the query language consists of familiar relational operators over streams of tuples, extended to specify joins between causally related events (with the happened-before operator), which enables some neat "pivot table" styles of analysis across data generated by different types of components and at different points in a request's lifetime.

Pivot Tracing favors specificity in return for low cost; removing the need to down-sample requests.

The paper also proposes another interesting twist to the conventional approach with the notion of *baggage*. With continuously executing queries running in-situ, where does the input data come from? How does information generated at one component (say, the name of the client application) reach a query running on a different component (say, a file system node that will join this name with a count of bytes read)? Baggage is a container for propagating the causally related "stream of tuples" in-band, along with the request itself. This is an intriguing design choice because the propagation and query processing costs are borne by the live system itself, and thus have to be managed carefully, but in return we have a flexible and powerful tool for interactive debugging of a complex, distributed system.

Although tracing systems have been around over 20 years, their use in production has only become mainstream in the last few years. Tracing support is now offered by most cloud providers to their customers, and there is an active open source community, defining standards such as OpenTracing (with baggage now part of the specification), OpenCensus, and OpenZipkin. Nevertheless, there is still a great deal of unrealized potential in tracing for sophisticated debugging and rich analytical insights to help manage complex distributed systems, and this thought-provoking paper makes a timely contribution to the conversation.

Rebecca Isaacs is a software engineer at Twitter in San Francisco, CA, USA.

The views here are the author's own, and do not reflect the views of Twitter.