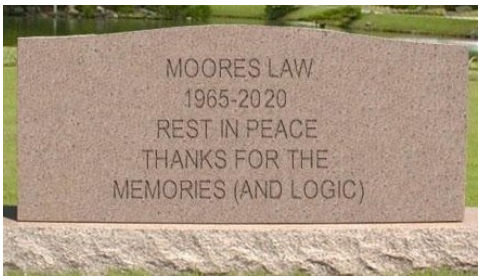
Metrics for Microservices

BY IAN PYE, CO-FOUNDER & PRINCIPAL ENGINEER · NOV 16, 2015

Time-series reporting for performance optimization

Once upon a time, life was simple. Programs ran in a single thread and performance was CPU bound. (I know this is a simplification, but bear with me as I try to make a point.) Using fewer instructions resulted in faster runtimes, and Moore's Law could be counted on to reduce the cost of each instruction with every new CPU generation.

Today, however, we live in scary times. Moore's law is effectively over. Applications now have to scale horizontally to meet performance requirements. And any time you have to deal with something more than a single box can handle, you introduce a whole host of complications (network, NUMA, coordination, and serialization, to name just a few).



KDE handles over 10B flow records/day with a microservice architecture that's optimized using metrics.

Here at Kentik, our Kentik Detect service is powered by a multi-tenant big data datastore called Kentik Data Engine. KDE handles — on a daily basis — tens of billions of network flow records, ingestion of several TB of data, and many millions of sub-queries. To make it work at a cost that's far below any existing solution, we've had to start off smart, with an in-house backend that can keep up with the volume of data and queries. That means scaling horizontally, which involves a complex distributed system with a custom microservice architecture. And that leads us to metrics.

Instrumentation as a key requirement

When we designed KDE, one of our key enabling requirements was to have instrumentation for end-to-end of monitoring of service delivery. My co-founder Avi Freedman's experience at Akamai had already shown the value of having, for internal and customer use, functionality along the lines of Akamai Query. Query, a combo real-time datastore and query engine, allows Akamai operations to monitor the health of their services and to "reach in" and query application components. Akamai uses it to support operational, performance, security, and efficiency use cases.

We knew that a similar level of support for holistic end-to-end monitoring of the constituent components of user performance — application, application environment, and host as well as data center, WAN, and Internet networks — would be needed to enable KDE to ingest trillions of items, to return timely SQL query results, and to be compatible with the architecture that we've built for our customers. So it was critical to instrument every component leading to, around, and within our data engine. Done right, there is almost no such thing as too much instrumentation!

Done right, there is almost no such thing as too much instrumentation.

The life of a query

Consider briefly the life of a query in Kentik Detect. Using a psql client, our RESTful API, or our portal, a user accesses the KDE's PostgreSQL-based frontend and runs the following query:

```
SELECT src_as, sum(in_pkts) as f_sum_in_pkts FROM all_devices WHERE i_start_time >= now() - interval'5 min'
GROUP BY src_as ORDER BY f_sum_in_pkts DESC LIMIT 10
```

In English, this is saying to give me the top 10 AS numbers sending traffic to my network over the last 5 min. Simple, right? Not so fast. All of the underlying data lives on distributed nodes across a cluster. To make the Kentik Detect SaaS offering work at scale on a multi-tenant platform we needed to enable query rate-limiting and subquery result memorization. That means that a lot of work has to happen under the hood, all implemented as separate but cooperating services.

Here's what happens behind the scenes:

- The query gets first validated against the schema in PostgreSQL of the all_devices table (a pseudo-table merge of the network data from all of a customer's devices sending flow records to KDE).
- From here it is sent to one of our middleware processes, chosen via Least Recently Used (LRU) from the pool of possibilities.
- The middleware:
  – Runs a bunch of additional validations (Who is logged in? Does the user have access to the requested data? Etc...);
  – Breaks the main query into five subqueries, one for each time unit covered by the main query (data is stored in 1 minute buckets).
  – Further breaks the subqueries into one sub-subquery per device. Assuming that the user has 10 devices, this leaves us with 5 * 10 = 50 sub-subqueries (jobs), all of which must be fulfilled before the user sees anything.
- The master next connects to our metadata service and, for each job, selects one worker to service the job. Workers are processes that run on our storage nodes.
- The chosen worker gets a job, validates again that it has local data to run on, and, if valid, enqueues the job for processing.
- Another thread will pick up this job when it is ready to run (the scheduling algorithm for which will be the subject of a future blog post).

- The job is run, result checked for errors, and sent back to PG via the middleware process that requested it.
- Once all 50 jobs return to PostgreSQL:
  – The data is de-serialized, turned into PG tuples, and sorted;
  – LIMIT and GROUP BY are applied
  – The top 10 results are displayed to the user.

The above is actually a simplified version of what really happens, because I cut out a lot of the annoying corner/failure cases that we have to handle!

We've refined our production system to the point where the steps above all happen in a few hundred milliseconds. But things didn't start out that optimized. So given all these moving parts, how did we figure out where performance was bottle-necking? Metrics.

Health checks and series metrics

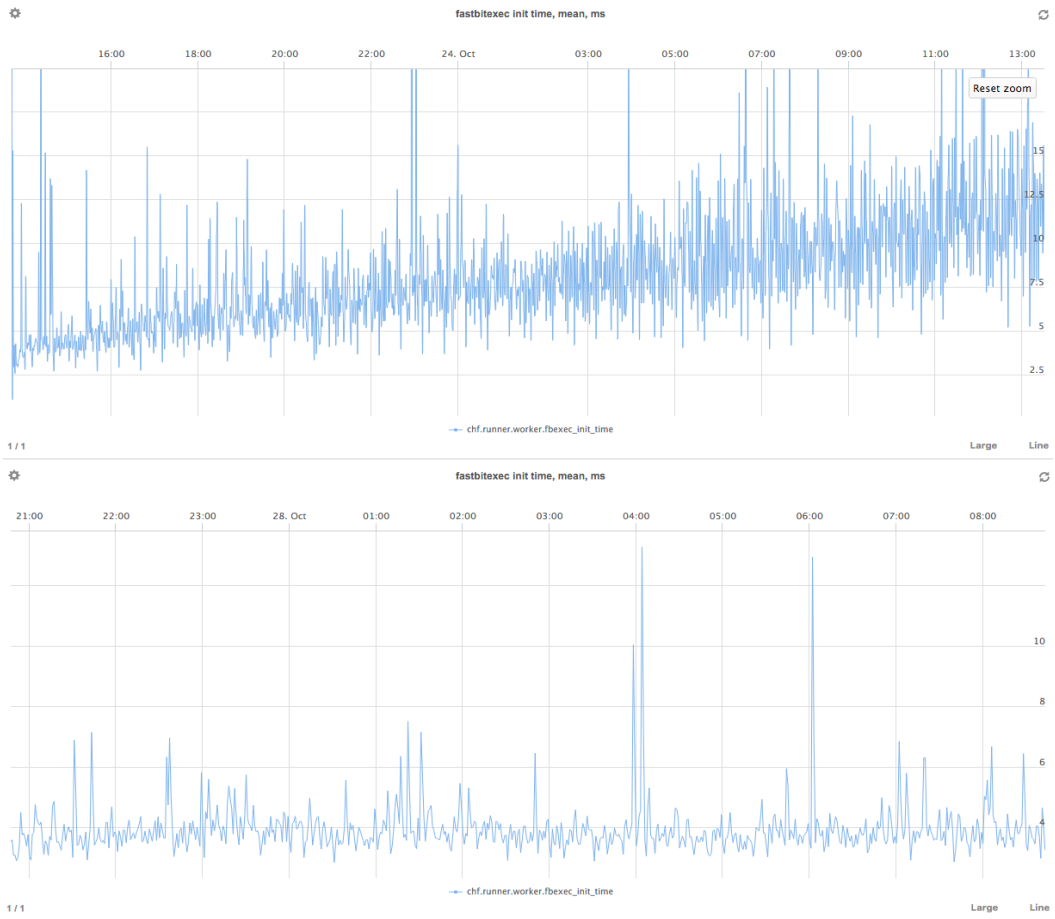
As a first pass, all of our binaries use a healthcheck library. This allows us to connect to a local socket and get a static view of what's going on at any given time, along with process version and build information. For example, here's some sample output from an alert service:

```
vagrant@chdev:~$ nc 127.0.0.1 10001
GOOD
Alert System: 20ce3d935b6988a15b7a6661c8b6198bd1afe419 Built on Linux 3.16.0-4-amd64 x86_64 [go version go1.5
linux/amd64] Debian GNU/Linux 8.1 (jessie) (Thu Oct 29 01:51:25 UTC 2015)
————————-
1013 In Q: 01013 16 | V3 Base Q: 0
1013 16 | V3 Long Q: 0
1013 16 | V3 State Q: 0
1013 16 | V3 Rate Spot: 50.195457
1013 16 | V3 Rate Select Spot: 50.035864
```

Note that the system is GOOD, queue depth is 0, and the flow rate is 50 flows/sec, of which all are being selected for further processing. This is useful, but really only in a binary sense: things are either completely broken, or working fine. Queue depths are max, or 0. Very rarely are they reported as somewhere in the middle, which in practice is where things get interesting. We also get a rate for events/second, but is this a local max? A local min?

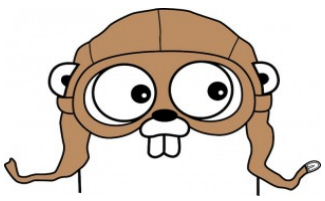
Graphed, time series metrics make obvious at a glance if new code is helping.

To dig deeper and get useful numbers, we need metrics reporting over time. Graphed, time series metrics make obvious at a glance if new code is helping. For example, consider this before-and-after image:



The graphs show the time in milliseconds needed to init a query worker. The top is before a fix, while the bottom version is after. Obvious, right? Up and to the right is not good, when you are talking latency. But just try confirming this without a centralized metric system. Internally, we are doing this with OpenTSDB to store the metrics, a PostgreSQL Foreign Data Wrapper used as an ops interface and inter-component glue, and Metrilyx as the front-end. In the future we may change to a different time series datastore for the "simple" time series metrics. But OpenTSDB, when enhanced with some "operationalization" diapers like the SQL interface, works well for now, and it has rich integrations that we can leverage for monitoring hosts and application components.

Metrics tags in Go



Our backend code is mostly written in the Go language. We heavily leverage Richard Crowley's excellent go metrics library (https://github.com/rcrowley/go-metrics), which is itself a port of Coda Hale's and Yammer's Java language library. The winning feature here is that Crowley's code exports directly to OpenTSDB (along with Graphite and several other backends), saving us the trouble of writing a connector. One addition we did make, however, is the ability to register custom tags for each metric (e.g. node name and service name).

The library offers five main types:

- **Counter** — This is what it sounds like: just a number that goes to MAX_INT and then rolls over. Not so useful,

we've found.

- **Gauge** — Similar to a Counter, but goes up and down. Reports a single number, for example workers remaining in a thread pool. Again, not particularly useful because it lacks a time component.
- **Meter** — Records a rate of events per second. Any time one is looking at a stream of events or jobs or processes, this is your friend.
- **Histogram** — A gauge over time. The champ of metrics. Records percentile values. This allows us to say that the min depth of our worker pool is Y, while 95th percentile is Z.
- **Timer** — Just like histogram, but optimized to take in duration of events. Using these, we track times for the max, min, 95th, 50th, etc. for each stage of our query pipeline .

Fewer surprises, happy customers

We run a microservice architecture, with components varying from microservice to "milliservice" in complexity. Every service is extensively instrumented, along with a whole host of server stats via the invaluable project tcollector. After every release, this setup allow us to compare before and after stats to see exactly what has changed. This makes for many fewer surprises, which makes for happy customers.

One more thing: because we write code in Go, we get to cheat. Go supports real time performance profiling (both memory usage and time spent in each function). Even better, you can connect your profiler to a running process via HTTP (https://golang.org/pkg/net/http/pprof/). Remember how all of our processes expose a health check port? They also expose a PPROF HTTP port. At no runtime cost unless actively profiling, every binary allows us to tap in at any time and see where it is spending its time. How to use this will be covered in yet another blog post.

Think some or all of this is fascinating? We're hiring!