Scaling to billions
Lessons learned from Circonus
What is Circonus?

- and why does it need to scale?
Circonus

- Telemetry collection
- Telemetry analysis
- Visualization
- Alerting

- SaaS... trillions of measurements
Architecture

- Distributed collection nodes
- Distributed aggregation nodes
- Distributed storage nodes
- Centralized message bus (for now, federate if needed)
Distributed Collection

- It all starts with Reconnoiter
  https://github.com/circonus-labs/reconnoiter

- This is, by its design, distributed.
- It is hard to collect billions of measurements per day on a single node.
Jobs are run

- They collect data and bundle it into messages (protobufs)
- A single message may have one to thousands of measurements.
- Data can be requested (pulled) by Reconnoiter
- Data can be received (puhed) into Reconnoiter
- Uses jlog ([https://github.com/omniti-labs/jlog](https://github.com/omniti-labs/jlog)) to store and forward the telemetry data.
- C and lua (LuaJIT).
Aggregation

- In the beginning, we started with a single aggregator receiving all messages.
- Redundancy was simple, but scale was not.
- It is not that difficult to route billions of messages per day on a single node.
Storage

- We started with Postgres
- Great database, but our time-series data was not efficiently stored in a relational database.
Column-store hack on Postgres

- We altered the schema.
- Instead of one measurement per row...
  - 1440 measurements in array in a single row
  - one row per day
Data volume grew

- Our data volume grew and we federated streams of data across pairs of redundant Postgres databases.
- This would scale as needed.
- Had unreasonable management burden.
Safety first...

- How to switch from one message queueing technology to another?
- Cut-over always has casualties.
- Solution 1: partial cut-over
- Solution 2: concurrent operations
We built “Snowth”

- We realized all our measurement storage operations were commutative.
- We designed distributed, consistent-hashed telemetry storage system with no single point of failure.
Availability Zone 1

Availability Zone 2

o1
A real ring

❖ Keep it simple, stupid.
❖ We actually don’t do split AZ
Rethinking it all
Time & Safety
As needs increased...

- We added computation support into the database.
  - allowing simple “cheap” computation to be “moved” to the data
  - allowing data to be moved to complex “expensive” computation
Real-time Analysis

- Real-time (or soft real-time) requires low latency, high-throughput systems.
- We used RabbitMQ.
- Things worked well until they broke.
Message Sizes Matter

- Benchmarks suck: they never resemble your use-case
- MQ benchmarks burned me.
- Our message sizes are between 0.2k and 6k.
- It turns out many MQ benchmarks are at 0k. *WHAT?!*
RabbitMQ falls

- Ar around 50k message per second at our message sizes, RabbitMQ failed.
- Worse, it did not fail gracefully.
- Another example of a product that works well... until it is worthless.
Soul-searching lead to Fq.

- We looked and looked.
- Kafka led the pack, but still...
- Existing solutions didn’t meet our use-case.
- We built Fq:
  https://github.com/postwait/fq
Subscribers can specify queue semantics:
- public (multiple competing subscribers), private
- block (pause sender flow control), drop
- backlog
- transient, permanent
- memory, file-backed
- many millions of messages (gigabits) per second
Safety first...

- How to switch from one message queueing technology to another?
- Cut-over always has casualties.
- Solution 1: partial cut-over
- Solution 2: concurrent operations
Duplicity

- “concurrent” deployment of message queues means one thing:
  - every consumer must support detection and handling of duplicate message delivery
  - there are three numbers in computing: 0, 1 and ∞.
谢谢

Thank You