Scaling to billions

Lessons learned fom Circonus

Theo Schlossnagle

@postwait

CEO Circonus

What is Circonus?

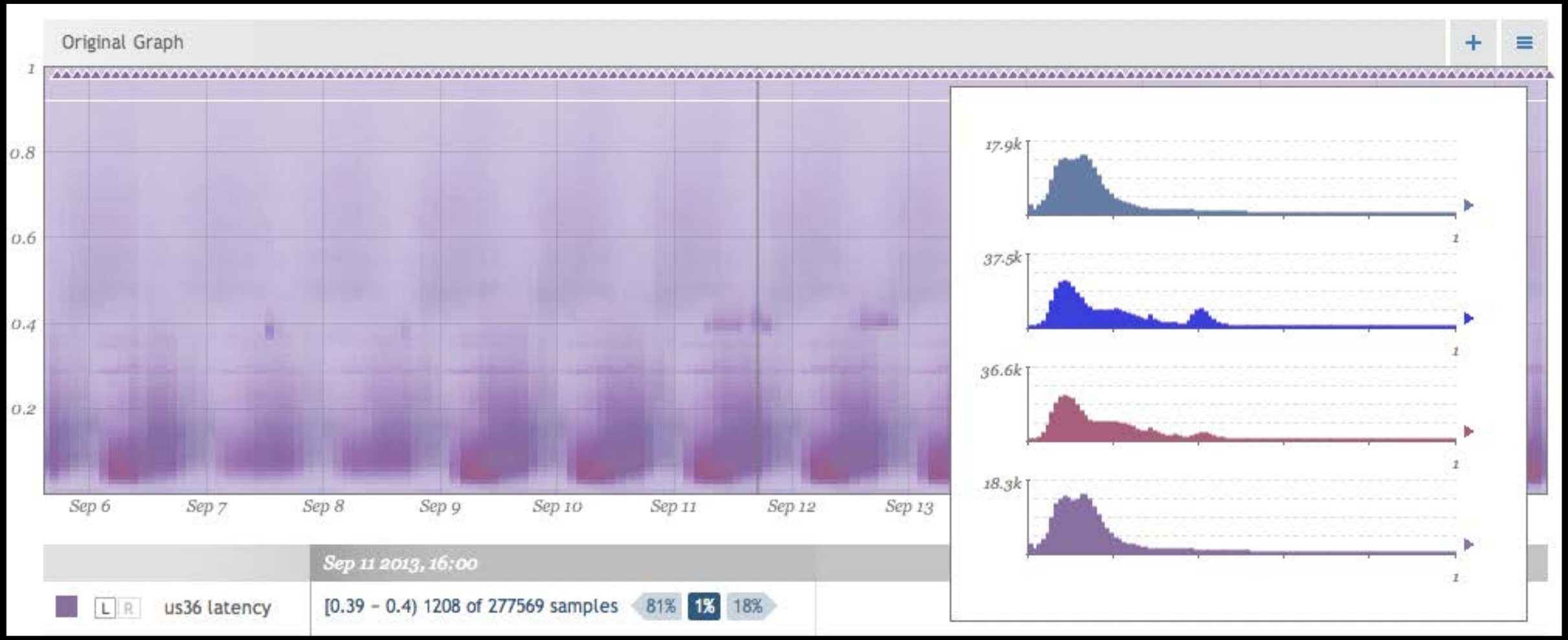
- and why does it need to scale?

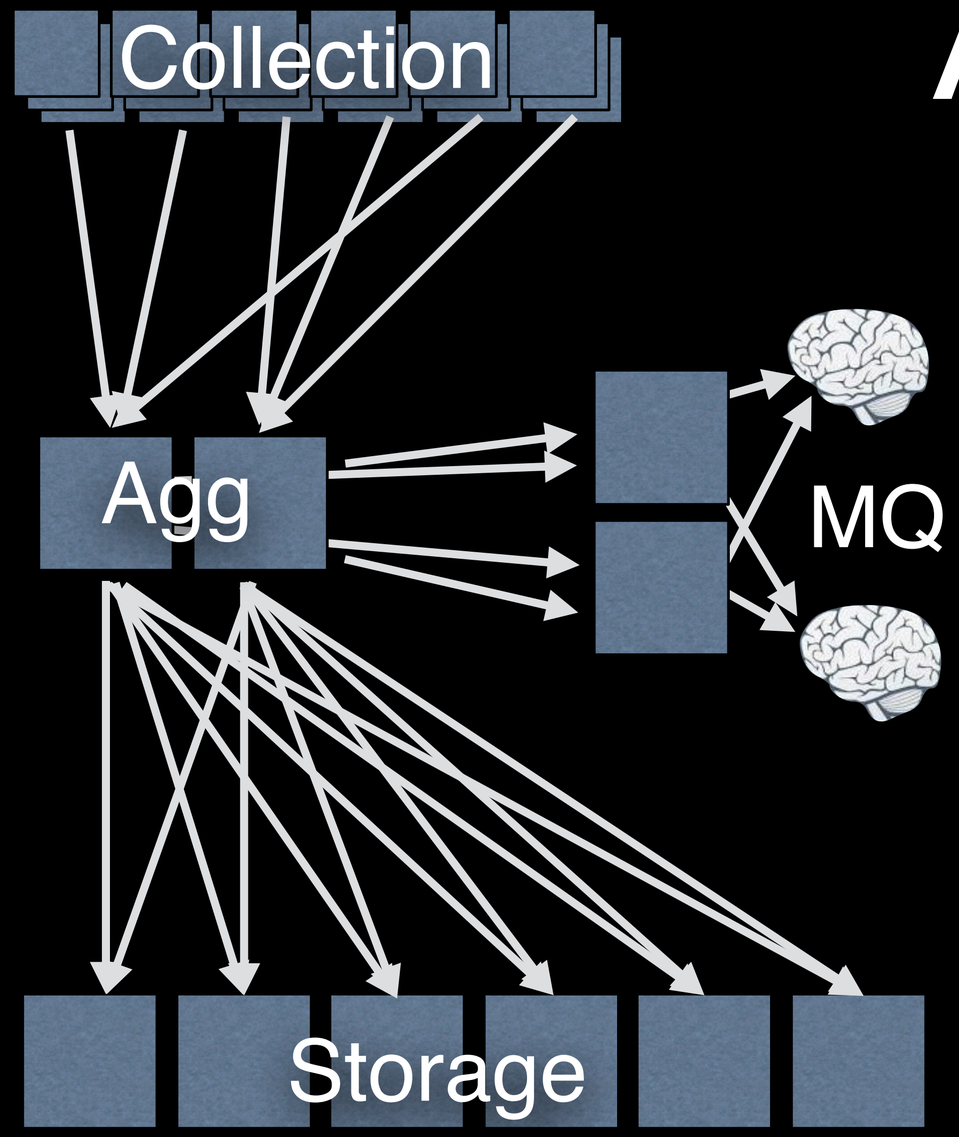
CIRCONUS

Circonus

- Telemetry collection
- Telemetry analysis
- Visualization
- Alerting

- SaaS… trillions of measurements


Original Graph
1
0.8
0.6
0.4
0.2
Sep 6
Sep 7
Sep 8
Sep 9
Sep 10
Sep 11
Sep 12
Sep 13
17.9k
1
37.5k
1
36.6k
1
18.3k
1
Sep 11 2013, 16:00
L R
us36 latency
[0.39 - 0.4) 1208 of 277569 samples
81%
1%
18%



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/reconnnoiter

- 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) 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

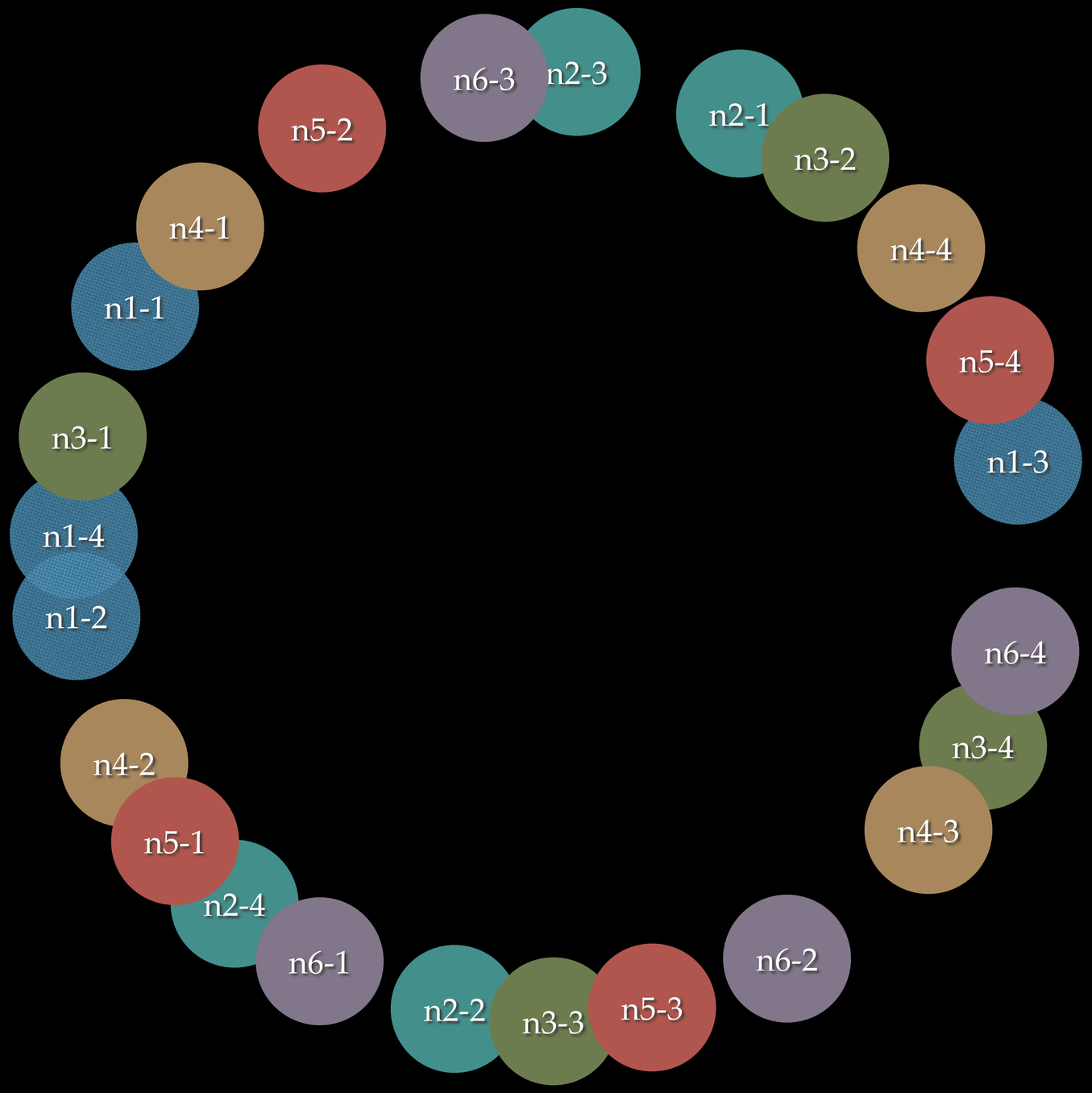
- Our data volume grew and we federated streams of data across pairs of redundant Postgres databases.
- This would scale as needed.
- Had unreasonable management burden.

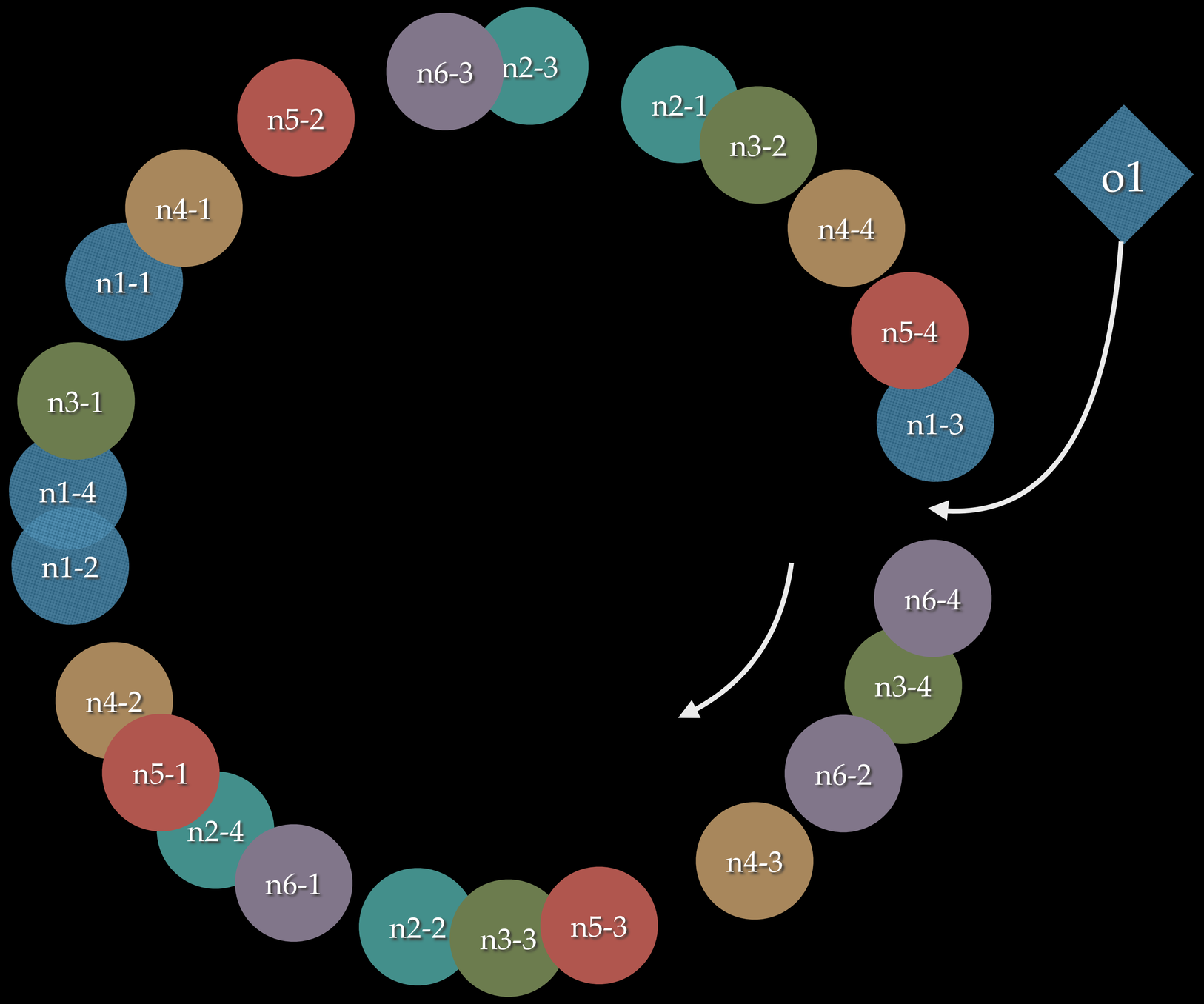
Safety first…

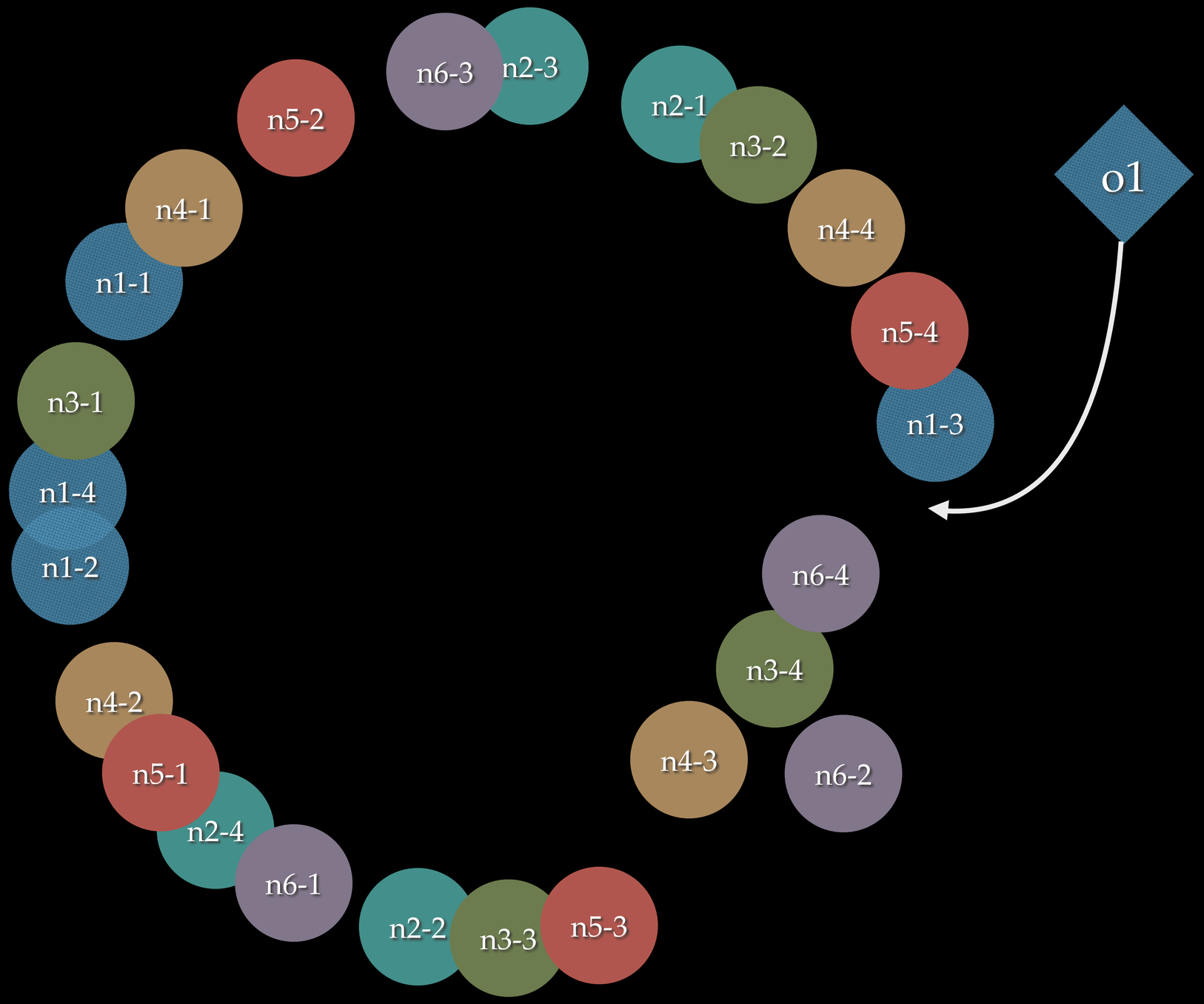
- How to switch from one message queueing technology to another?
- Cut-over always has casualties.
- ~~Solution 1: partial cut-over~~
- Solution 2: concurrent operations

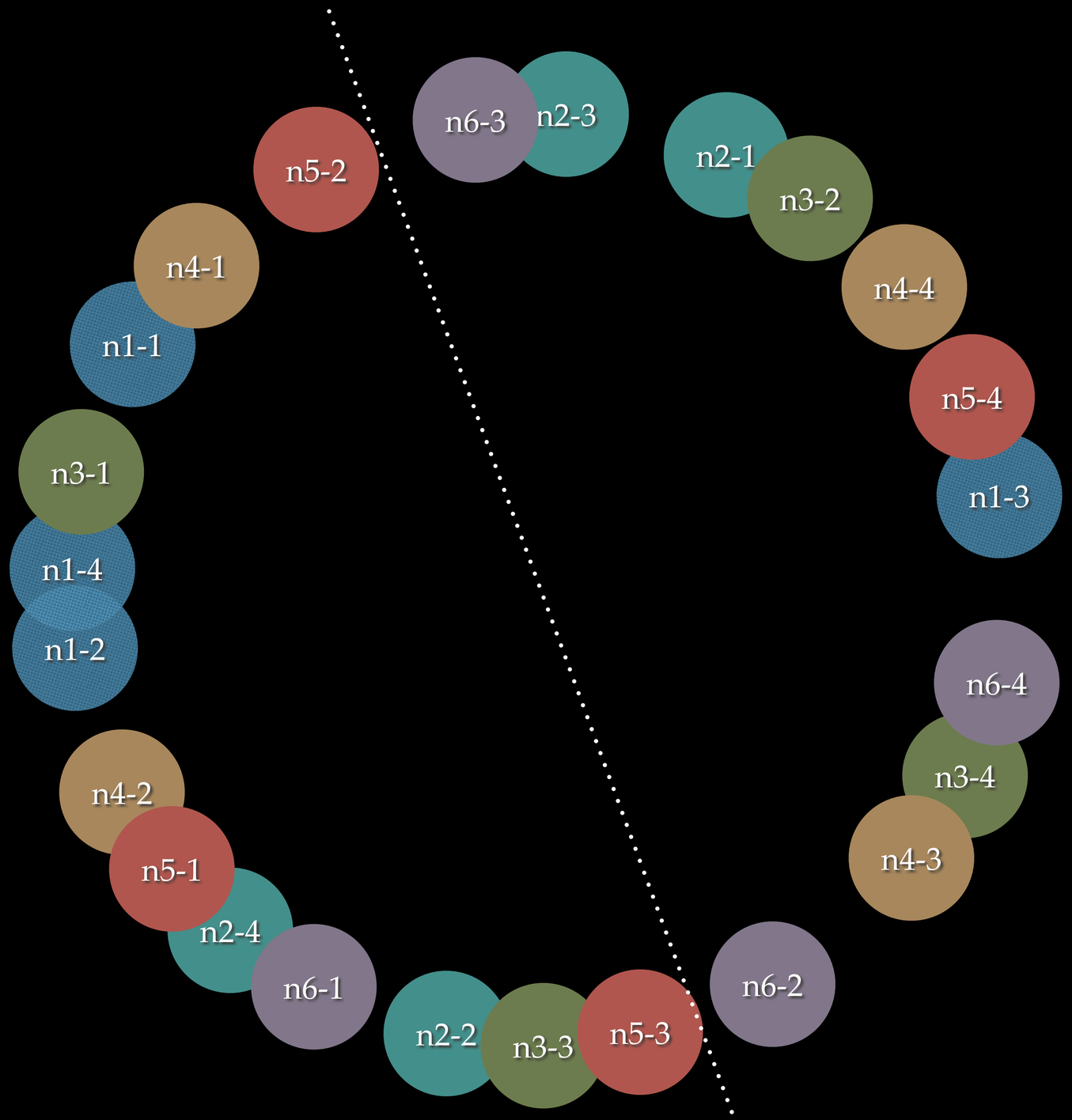
We built “Snowth”

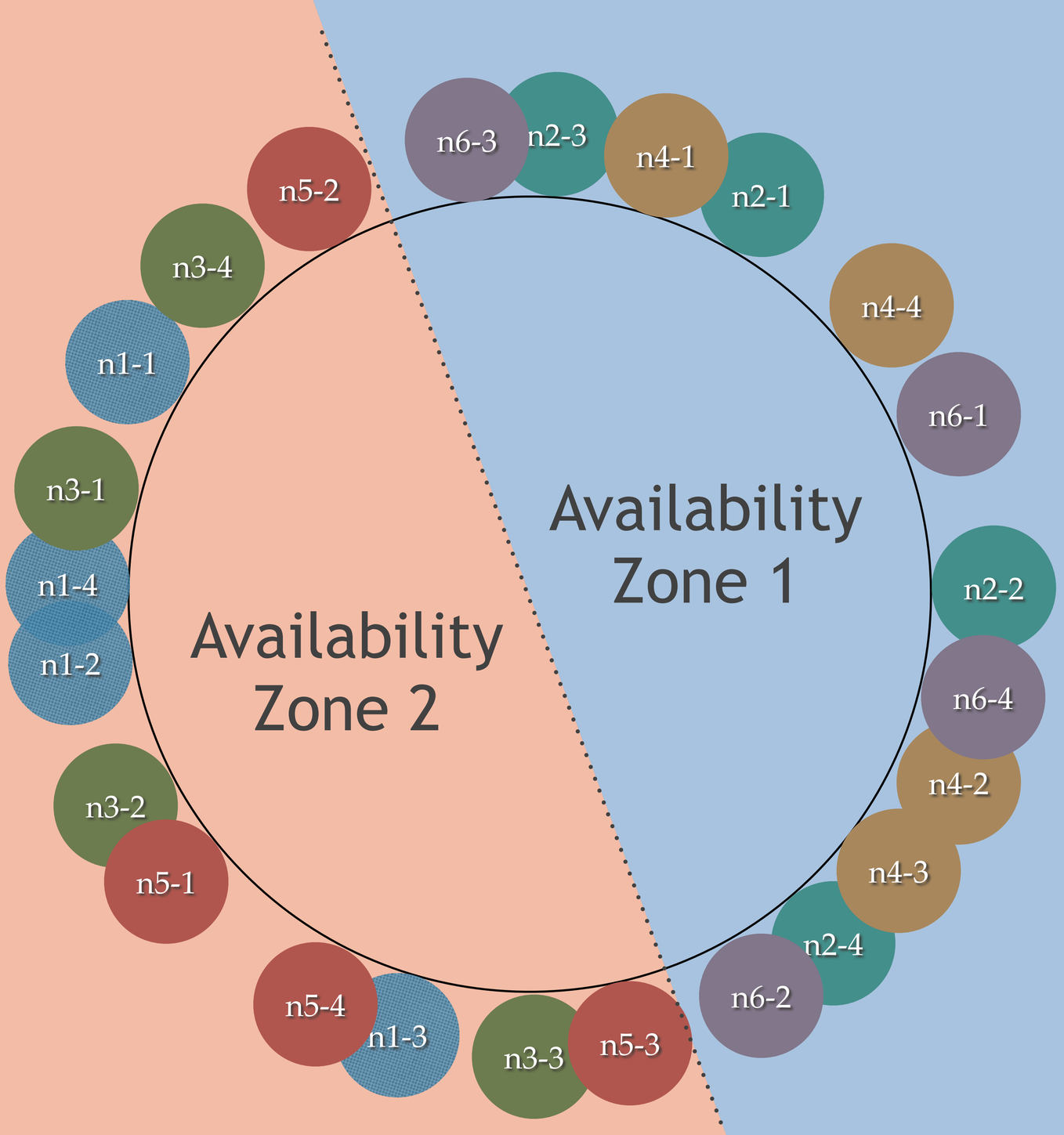
- We realized all our measurement storage operations were commutative.
- We designed distributed, consistent-hashed telemetry storage system with no single point of failure.

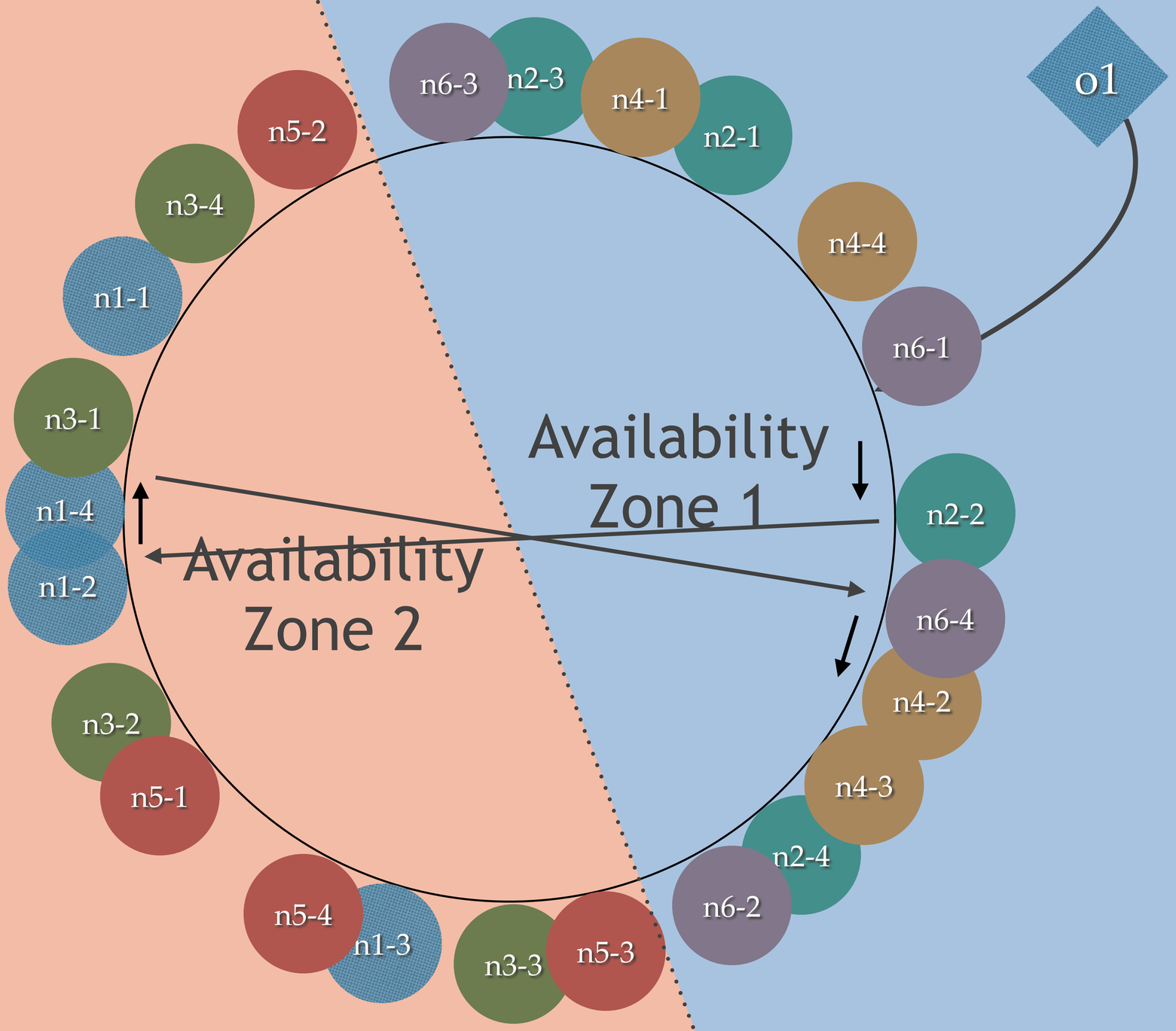

n6-3
n2-3
n5-2
n2-1
n3-2
n4-1
n4-4
n1-1
n5-4
n3-1
n1-3
n1-4
n1-2
n6-4
n3-4
n4-2
n4-3
n5-1
n2-4
n6-1
n6-2
n2-2
n3-3
n5-3

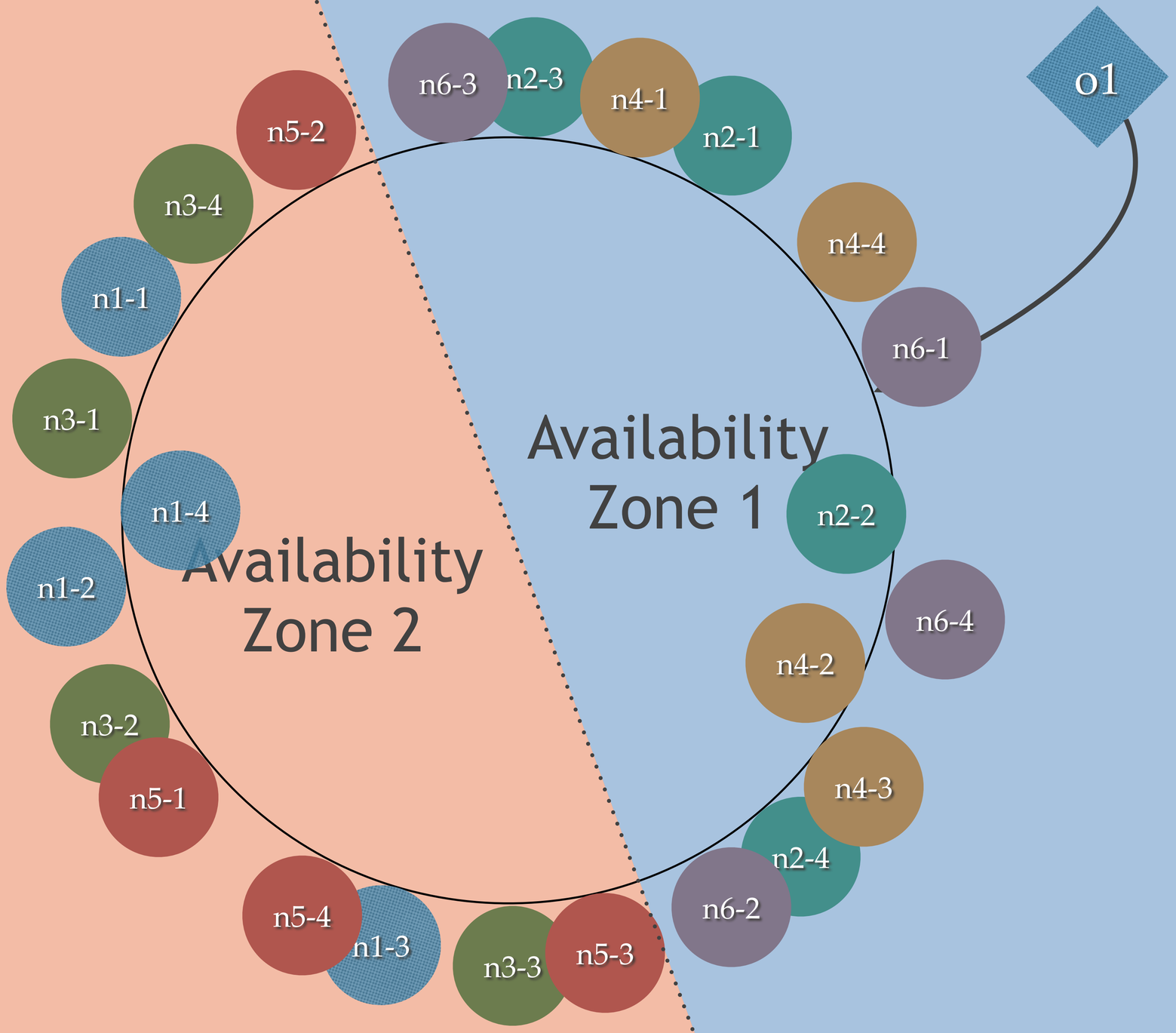

o1
n1-1
n1-2
n1-3
n1-4
n2-1
n2-2
n2-3
n2-4
n3-1
n3-2
n3-3
n3-4
n4-1
n4-2
n4-3
n4-4
n5-1
n5-2
n5-3
n5-4
n6-1
n6-2
n6-3
n6-4


n6-3
n2-3
n5-2
n2-1
n3-2
o1
n4-1
n4-4
n1-1
n5-4
n3-1
n1-3
n1-4
n1-2
n6-4
n3-4
n4-2
n4-3
n6-2
n5-1
n2-4
n6-1
n2-2
n3-3
n5-3


n6-3
n2-3
n5-2
n2-1
n3-2
n4-1
n4-4
n1-1
n5-4
n3-1
n1-3
n1-4
n1-2
n6-4
n3-4
n4-2
n4-3
n5-1
n2-4
n6-1
n6-2
n2-2
n3-3
n5-3


Availability Zone 1
Availability Zone 2
n6-3
n2-3
n4-1
n2-1
n4-4
n6-1
n2-2
n6-4
n4-2
n4-3
n2-4
n6-2
n5-3
n3-3
n1-3
n5-4
n5-1
n3-2
n1-2
n1-4
n3-1
n1-1
n3-4
n5-2


o1
n6-3
n2-3
n4-1
n2-1
n5-2
n3-4
n4-4
n1-1
n6-1
n3-1
Availability
Zone 1
n1-4
n2-2
Availability
Zone 2
n1-2
n6-4
n4-2
n3-2
n4-3
n5-1
n2-4
n6-2
n5-4
n1-3
n3-3
n5-3


o1
n6-3
n2-3
n4-1
n2-1
n5-2
n3-4
n4-4
n1-1
n6-1
n3-1
Availability
Zone 1
n1-4
n2-2
Availability
Zone 2
n1-2
n6-4
n4-2
n3-2
n4-3
n5-1
n2-4
n6-2
n5-4
n1-3
n3-3
n5-3

Circonus Snowth: 192.168.13.70:8112

	ID	Weight	Address	Coverage
	b7cc243f-85fa-4031-b69e-d6b9ed537093	85	192.168.13.70:8112	14.71%
	696fc43d-4cc8-4209-97da-f2ede7dcbc45	85	192.168.13.71:8112	15.30%
	b05f32f2-852b-42b9-921f-14de62b536cc	85	192.168.13.72:8112	16.32%
	36e8df00-2739-4c3a-80a1-a0aa9b29a44d	85	192.168.13.73:8112	13.82%
	bcd2d723-b936-416d-ba08-99d089b61376	85	192.168.13.74:8112	20.79%
	2b9f654f-40dc-447f-bc93-6f2c3818ef67	85	192.168.13.75:8112	19.07%

Locate Metrics

Check UUID

UUID

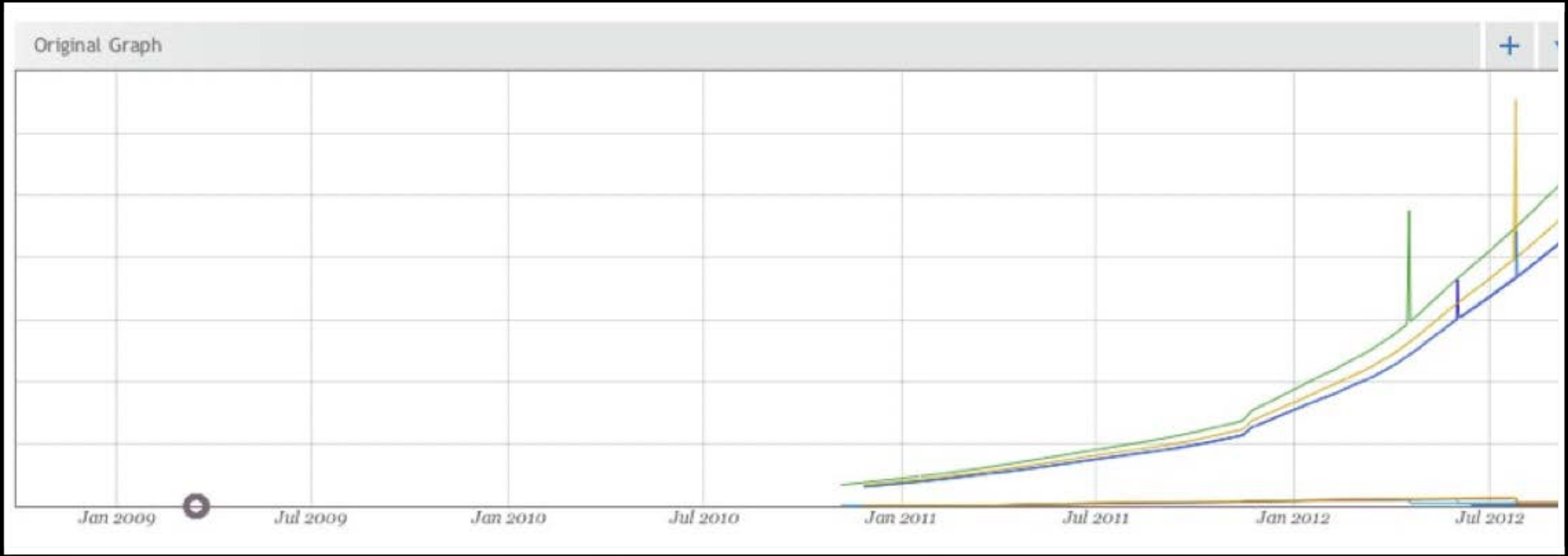
Metric Name

metric

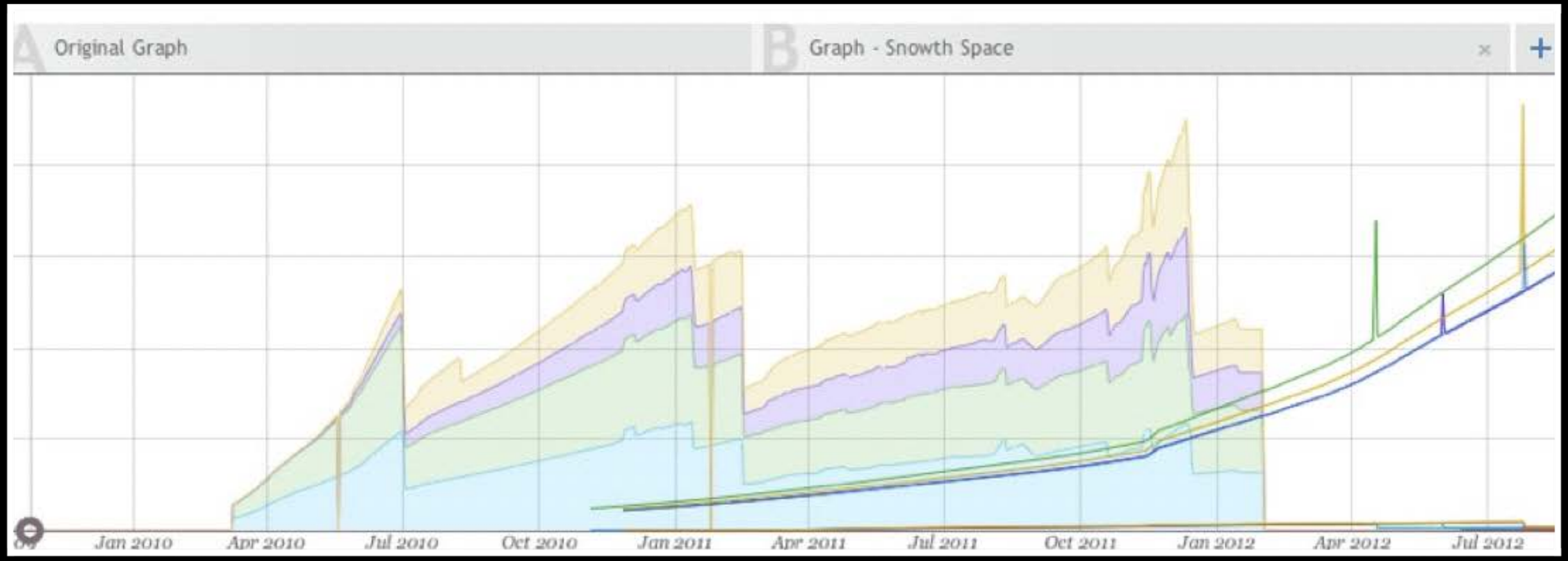
A real ring

- Keep it simple, stupid.
- We actually don’t do split AZ

Rethinking it all



Time & Safety


Original Graph
Graph - Snowth Space
Jan 2010
Apr 2010
Jul 2010
Oct 2010
Jan 2011
Apr 2011
Jul 2011
Oct 2011
Jan 2012
Apr 2012
Jul 2012

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

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 ∞.


O'REILLY
Velocity
China 2013
Web性能与运维大会
Beijing, China
Aug 20-21, 2013
velocity.oreilly.com.cn

谢谢

Thank You