Distributed Log-Processing Design Workshop

Introduction

SREcon Americas
Santa Clara, CA March 2018

Laura Nolan, Phillip Tischler, Salim Virji
Site Reliability Engineers, Google

Goals

Design a distributed log-processing system, going from the problem specification down to a bill of materials.

Perfect solution not required, "back-of-the-envelope"-style of reasoning.

Laptop not required.

You'll work in groups, assisted by facilitators.

Most important thing: have fun tackling a technical problem together. ☺ □

Agenda

09:05 – 09:10	Introduction
09:10 – 09:45	Considerations in Designing Distributed Systems
09:45 – 09:50	Handout distribution, break out in groups
09:50 – 10:00	Problem statement
10:00 – 10:30	Hands-on design activity (Architecture)
10:30 – 11:00	Break
11:00 – 11:40	Hands-on design activity (Provisioning)
11:40 – 12:10	Present example solution
12:10 – 12:30	Present final solution, discussion

Distributed Log-Processing Design Workshop

Considerations in Designing Distributed Systems

SREcon Americas

Santa Clara, CA March 2018

Topics

Architectural Considerations

Handling Failure

Managing Consistency and State

Source: Jim Ehle, Wikimedia, Creative Commons

Architectural Considerations

Gather requirements

It’s critical to understand data freshness needs.

You must try to understand current and future scale.

Can we treat some data differently than other?

Google

Source: MegLearner on Pixabay, Public Domain

What’s my scale?

How much data now and in the future?

Where does the data need to live?

How many copies to I need?

In what dimensions do I scale and which dominates?

Let your Service Level Objectives (SLOs) play a major role in your design.

Google™

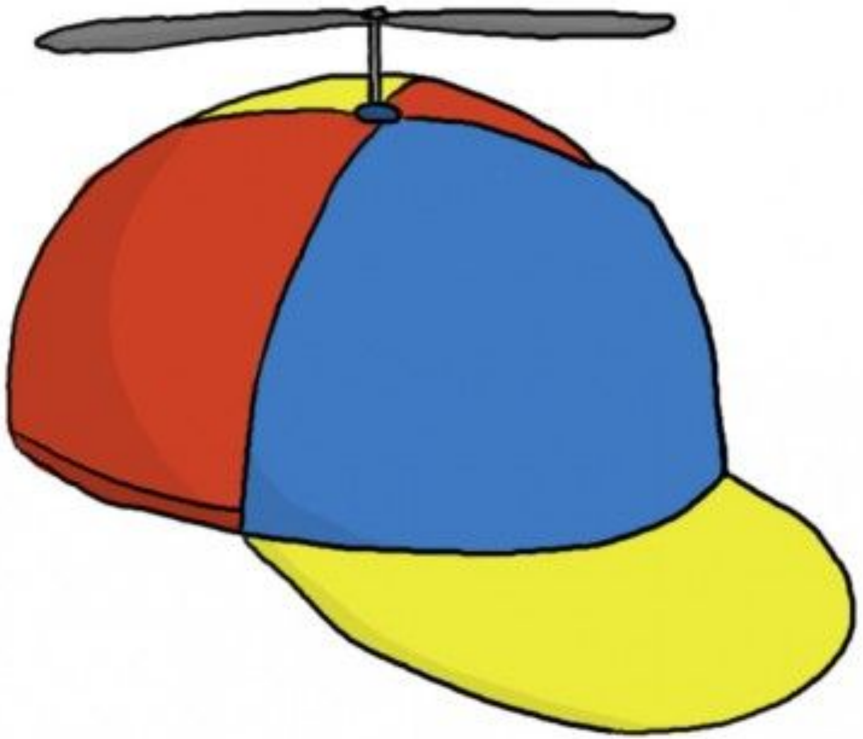
Source: Thomas Galvez, Creative Commons

CAP conjecture

Can't be consistent, available and partition tolerant all at the same time.

You really want partition tolerance.

Figure out which of your data needs to be available and which consistent.



Source: Nemosandman, Wikimedia, Creative Commons

Handling Failure

Failure domains

A set of things that might experience correlated failure

- A single process
- A machine
- A rack
- A datacenter
- Servers on the same software
- A global configuration system

Source: Andrew Wilkinson

Decouple your servers

Spread responsibilities across multiple processes

Watch out for global config pushes.

Canary changes and pause before global rollout.

Don't depend on one backend.

Keep serving if configs corrupt or fail to push.

Source: dschwen , Wikimedia, Creative Commons

Use multiple datacenters

Raises all the problems of distributed consensus.

But that’s ok, there are solutions.

Works well with load balancing.

Less well with cross-DC work.

Source: Robert.Harker, Wikimedia, Creative Commons

Be N+2

Plan to have one unit out for upgrade and survive another failing.

Make sure that each unit can handle the extra load.

Don’t make any one unit too large.

Try to make units interchangeable clones.
("Cattle, not pets")

Google

Source: Nick Royer, Flickr, Creative Commons

Managing Consistency and State

Try not to keep state

Stateless servers are easy.

Try to make as much of your system stateless as possible.

The best state is no state.

Source: pixabay, public domain

Shard it

Partition your data on something in the request (e.g. userid) into N buckets. Stateless but fragile in face of more servers. Fragile if server dies.

Have N*x*M shards assigned to M servers (with replicas). Reliable but needs coordination.

We can balance load at cost of needing to lookup shard location.

Google

Source: Steven Depolo, Flickr, Creative Commons

Distributed consensus protocols

Agrees on one consistent result between a collection of unreliable processes.

Requires a majority of processes to remain up and contactable.

Best known protocol is Paxos, others are ZAB (used in Zookeeper) and RAFT.

Individual replicas may be slow to converge (CAP).

Source: Wikimedia, public domain.

From distributed consensus:

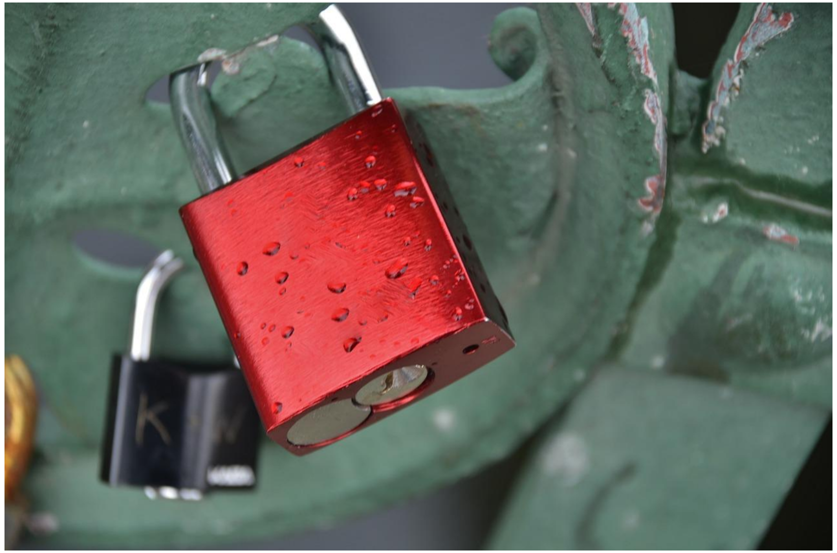
We can build distributed locking and slow filestores

From that we can build master election

From that we can make sure that only one of our replicas owns a shard.

We can also make sure we can find the replicas.

Overall, great for metadata, not so much for rest.



Source: pixabay, public domain

Distributed Log-Processing Design Workshop

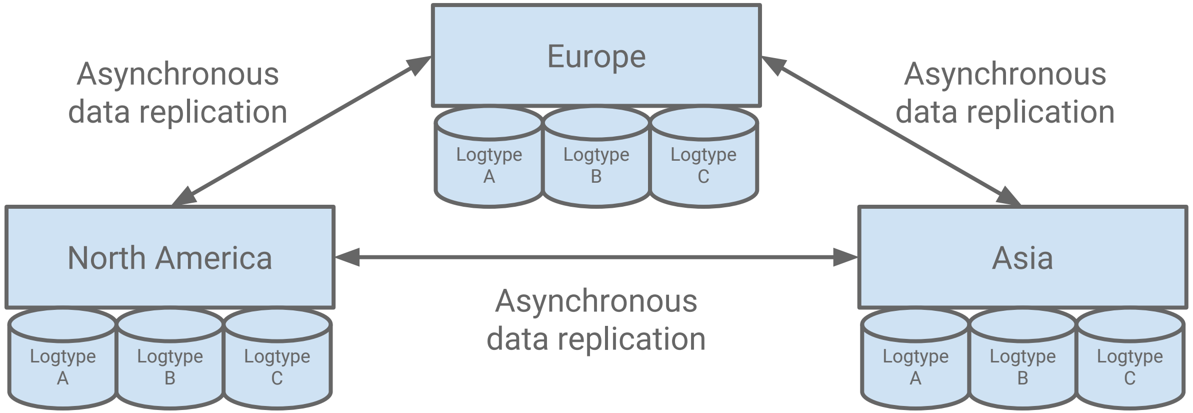
Problem Statement

SREcon Americas

Santa Clara, CA March 2018

Context

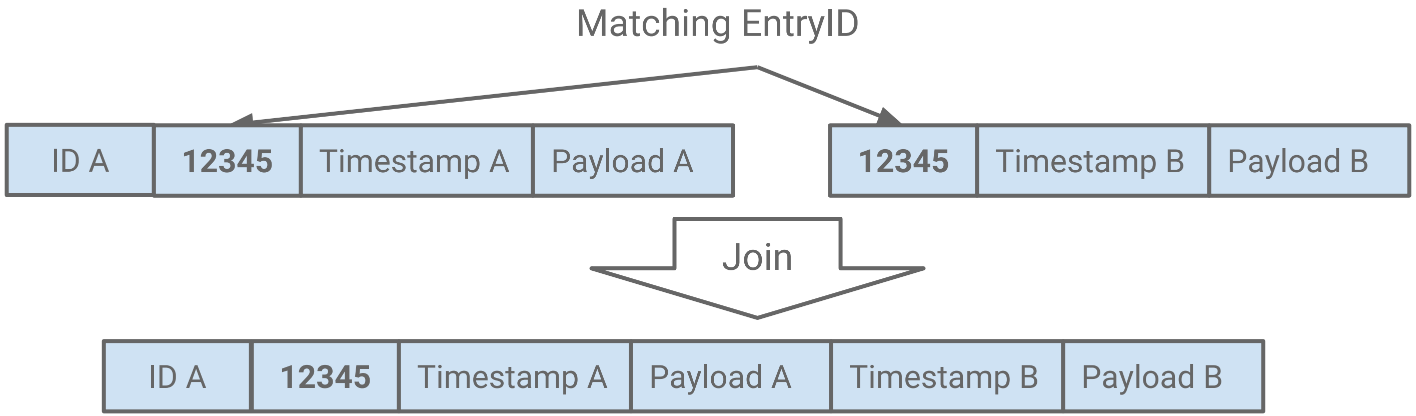
- We have 3 data centers (DC): Europe, North America, Asia.
- Services in each of them write log entries to a reliable distributed filesystem, one in each DC.
- Data is then asynchronously replicated to the other two.



Problem Statement

3 types of log entries: A, B, C; each log entry contains Entry ID, timestamp and payload.

Design a system which joins log entries of type A+B and A+C, based on a shared Entry ID.



Requirements

Reliability
The system should be reliable to failure of

- one DC
- one network link between any 2 DCs
- individual machines

Performance
95% of entries should be joined within 60 minutes from when it is first possible

Correctness
each joined pair **must** appear in the output at most once
each joined pair **should** appear in the output at least once
entries older than **10 days** can be ignored

Input Data Characteristics

- EntryID is 16B, timestamp is 8B (both already included in entry size)
- Logtype B and C can only be written after corresponding entries in Logtype A have been written

	Size per entry	Entries per day (globally)
Logtype A	**5kB**	approx. **10 billion (10^{10})**
Logtype B	**1kB**	approx. **250 million ($250*10^{6}$)**
Logtype C	**1kB**	approx. **50 million ($50*10^{6}$)**

Infrastructure

Filesystem

- Log entries are appended to files, which are wrapped at 1 GB
- Logs are stored on a reliable filesystem, distributed within the DC
- Files are replicated by using an asynchronous, reliable replication system with a delay of a few minutes
- Your system reads within the DC and writes within the DC.

Available types of machines (quantity unlimited)

- 24G RAM, 8 cores, 2x2TB hard drives, 1Gb ethernet
- 24G RAM, 8 cores, 2x1TB SSD, 1Gb ethernet

Suggestions on how to approach the problem

- Goal:
    - come up with an overall architecture and design, down to a bill of materials (# of machines needed)
- Reason about the data volume
- Focus first on the general architecture, then on the dimensioning

Google

Example Solution

Distributed Log-Processing Design Workshop

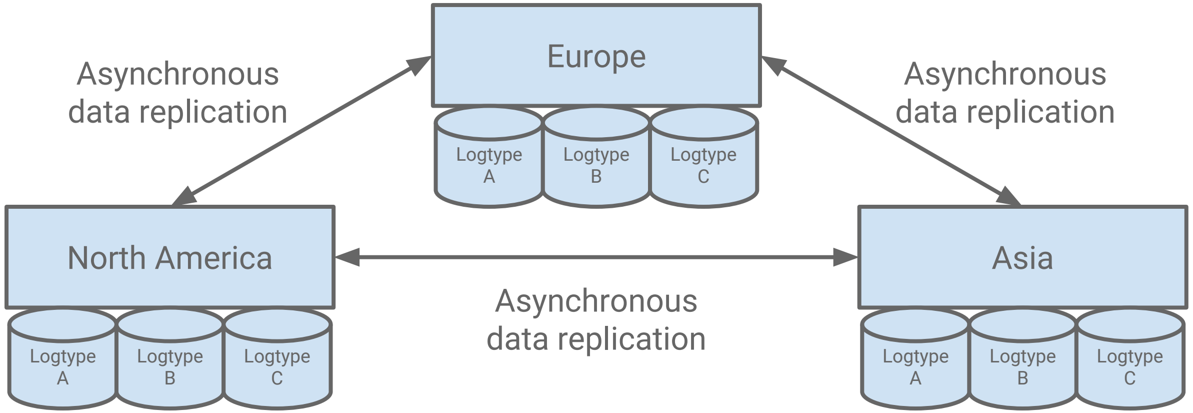
SREcon Americas

Santa Clara, CA March 2018

Problem Recap

High-level Constraints

- Join entries with matching ID across A+B, A+C
- Input files in 3 DCs
- The system must be reliable to an outage affecting a single DC or connection between any 2 DCs



Architecture of our example solution

Approach

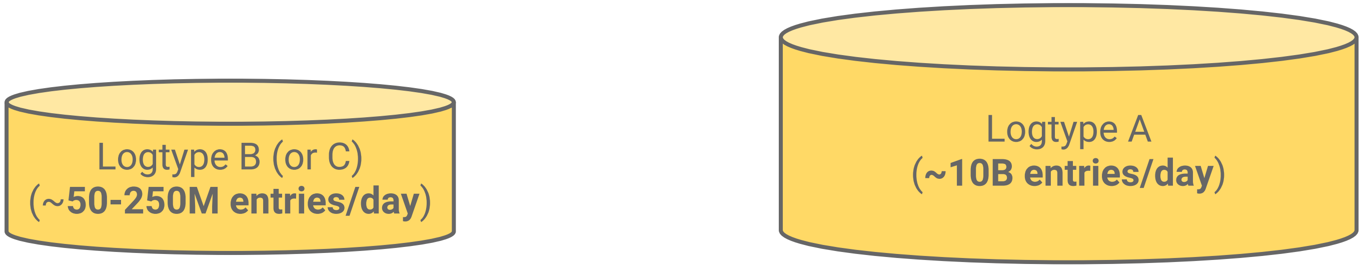
Naïve solution: 3 identical instances, each processing the same events.

Challenges:

- **correctness** - each joinable pair of log entries **must** appear at most once in the output
- **efficiency** - the system **should** avoid doing duplicate work

We'll address them after looking at the structure of one of these instances.

Reading input logs



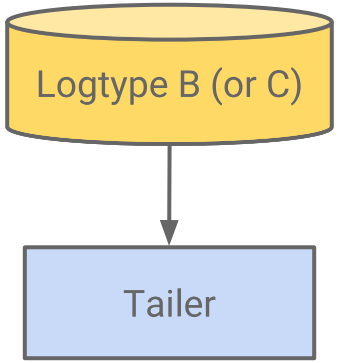
(drawing not to scale)

~100% of B/C log entries will be joined;

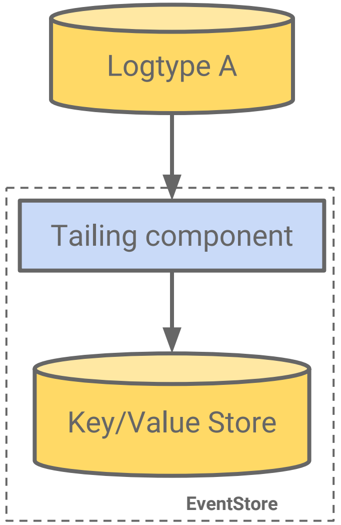
at most **~3%** of A log entries will be joined.

Tailer

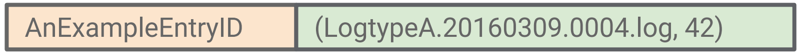
Tails open files, discovers new files.



EventStore Subsystem



Key: **EntryID**
Value: **(filename, offset)**

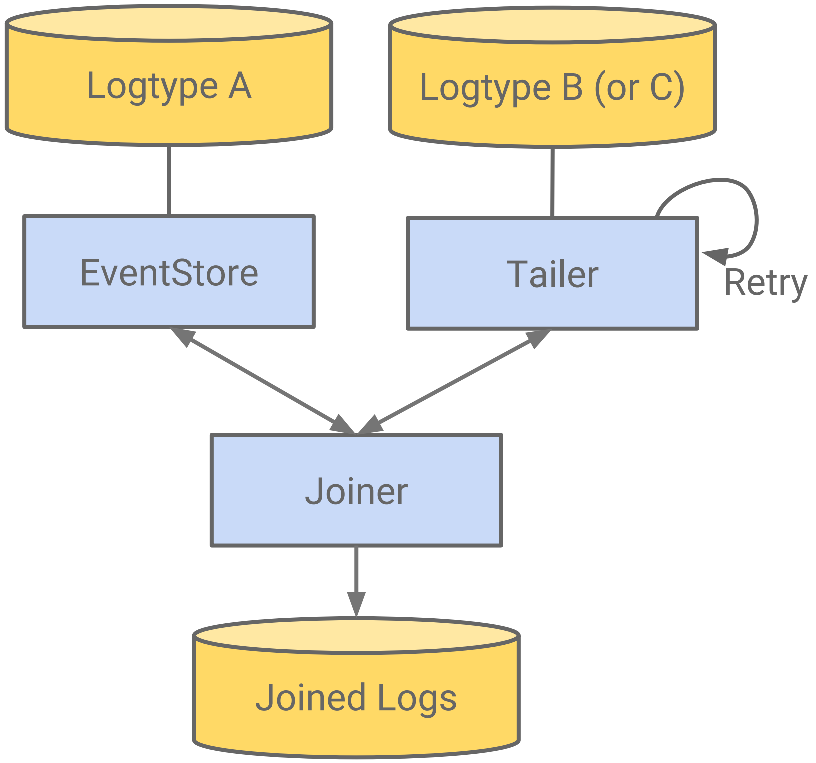


Joiner

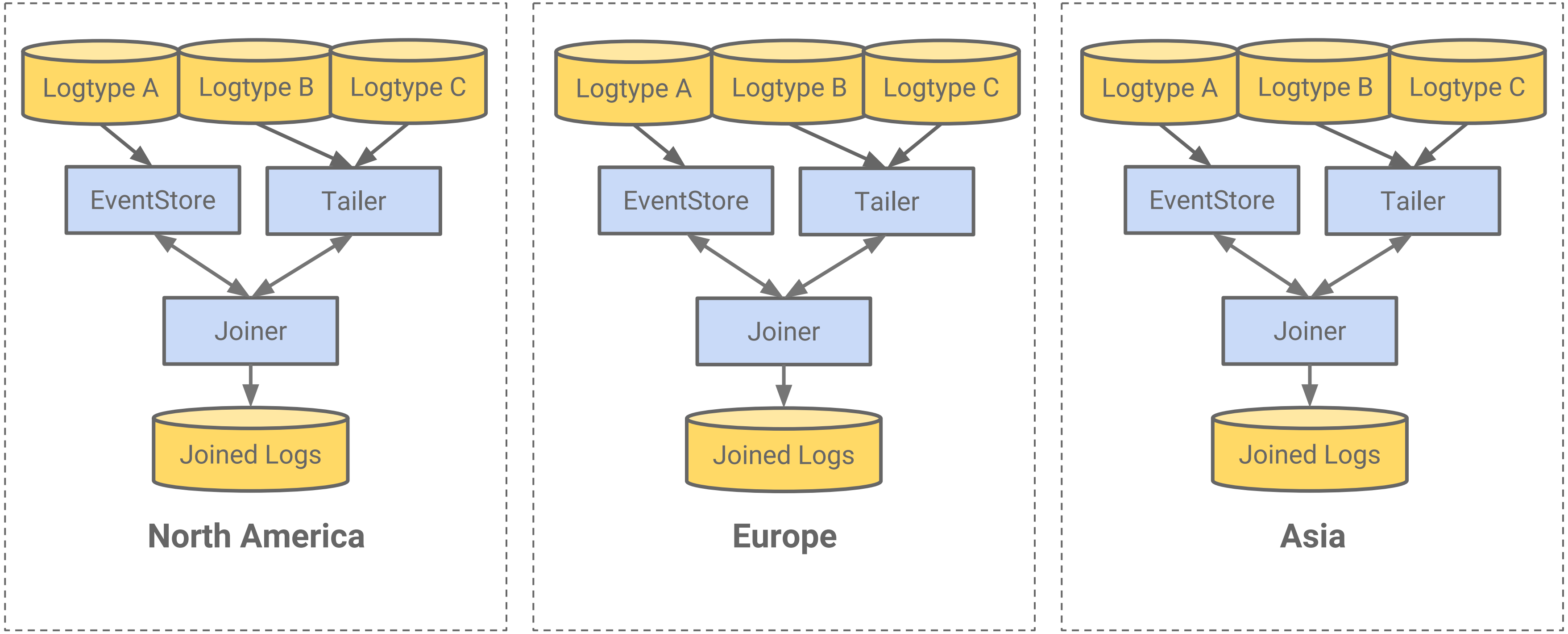
Replies to requests from the Tailer.

Once the Tailer sends a `Join` RPC, the Joiner:

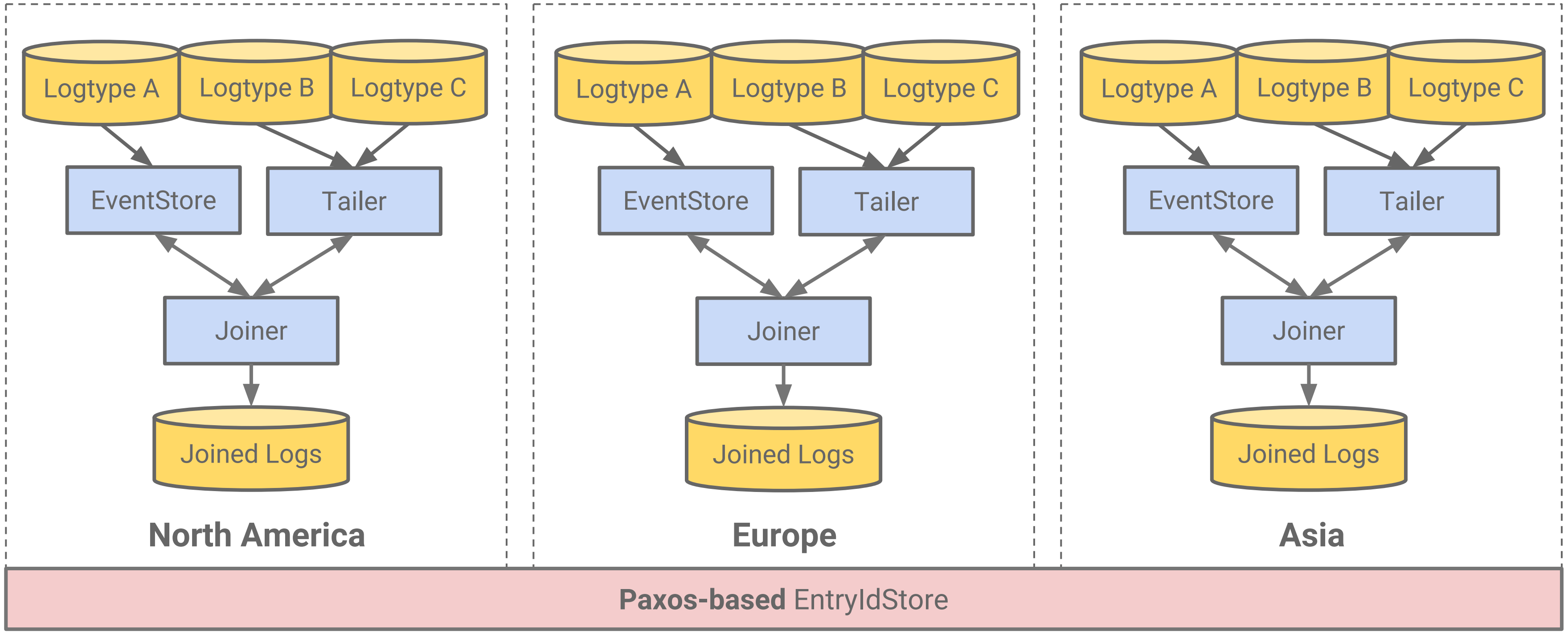
- fetches from EventStore the corresponding entry from A;
- **entry found**: writes the joined log entry to the output filesystem
- **entry not found**: send an error code back to the Tailer, that will back off and retry



Overall Architecture Diagram



Introducing Paxos-based distributed state



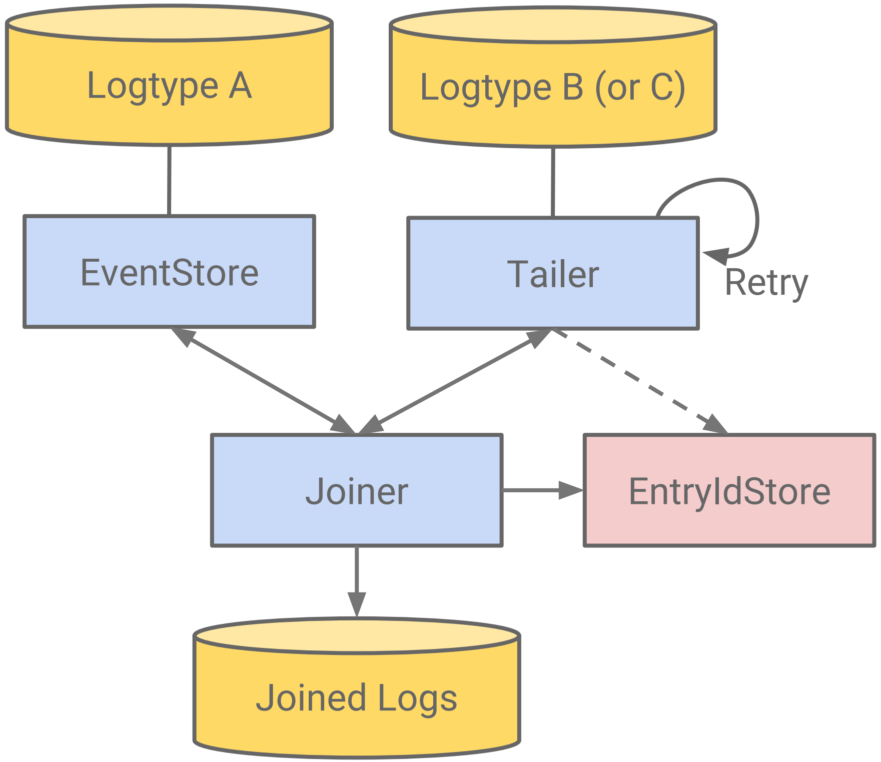
Role of EntryIdStore

Keeps in memory the joined EntryIDs.

Tailer can verify if the entry being read was already joined.

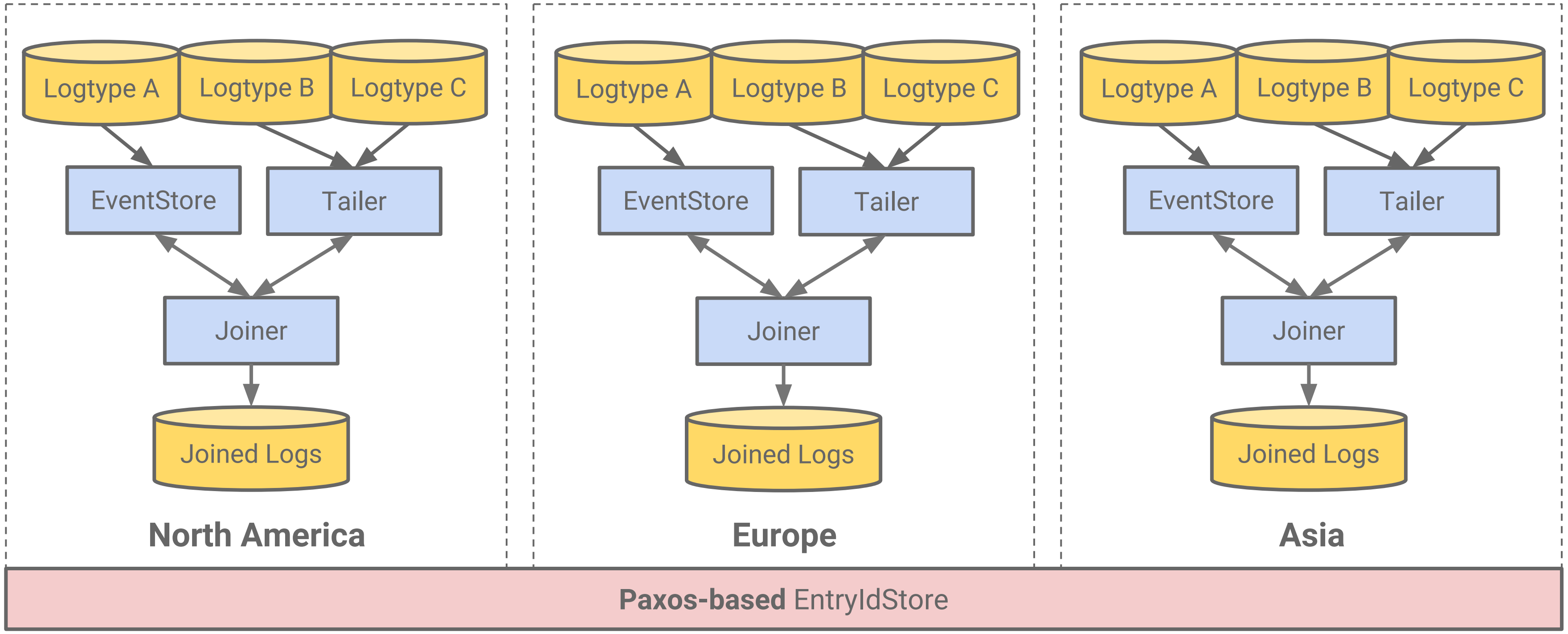
Joiner can write the EntryID of the joined entries.

This takes care of correctness and efficiency.

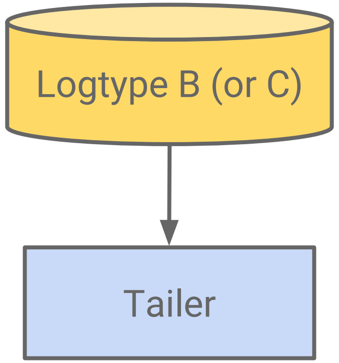


Dimensioning

Architecture Recap



Dimensioning Tailer



Bandwidth requirements:

250M entries/day * **1** kB/entry * **2** = **500** GB / day = **6200 KBps = 49.6 Mb/s**

Outgoing RPC rate: 250M entries/day = **2900 QPS**

Footprint: <u>1 machine</u>

Dimensioning EventStore: tailing component

Conceptually the same as the Tailer

Bandwidth: **10 billion** entries/day * **5** kB/entry = **50** TB/day = **570 MBps ~= 4.5 Gbps**

The network is going to be the bottleneck.

Footprint: 4.5 Gbps / 1 Gbps ~= 5 machines.

10 billion entries/day = **~120k** entries/second.

Would be good to batch writes to the key/value store.

Dimensioning EventStore: key/value store (storage)

Number of entries: **10** billion entries/day * **10** days = **100 billion** entries

Average size of a key/value pair:

$$\underbrace{\text{len(EntryID)}}_{\mathbf{16\ B}} + \underbrace{\text{len(filename)}}_{\mathbf{\sim100\ B}} + \underbrace{\text{size of offset}}_{\mathbf{4\ B}} = \mathbf{\sim150B}$$

100 billion entries/day * **150** B/entry → **~15 TB** space required

Footprint (in-memory): **15 TB / 20 GB** RAM/machine = 750 machines

Footprint (SSD): **15 TB / 2 TB** SSD/machine = 8 machines

Footprint (disk): **15 TB / 4 TB** disk/machine = 4 machines

Dimensioning EventStore: key/value store (latency/throughput)

QPS from clients: **(250+50)M** queries/day / **86400** s/day = **~3500** QPS

Payload size: 5 kB. Max QPS/machine network-wise: 1 Gb /5kB QPS = **25k QPS** (network is not a bottleneck)

Query latency (disk): 3ms (seek) + 0.05 ms (read) ~= 3 ms (theoretical max: ~300 QPS → 11 machines)

Query latency (SSD): 0.016 ms (random read) + 0.000625 ms (read) ~= 0.02 ms (max: ~8M QPS → 1 machine)

Query latency (memory): (less than SSD ☻, will hit other bottlenecks before that)

Dimensioning EventStore: key/value store (trade-offs)

	# Machines (storage)	# Machines (Latency)	Latency
Memory	750	1	0.01 ms
SSD	8	1	0.01 ms
Disk	4	11	3 ms

Choice: SSD (8 machines)

Dimensioning EventStore (recap)

Tailing component: 5 machines

Key-value store (SSD solution): 8 machines

Total: 13 machines

Dimensioning EntryIdStore (storage)

	Size per entry	Entries per day (globally)
Logtype A	**5kB**	approx. **10 billion (10^{10})**
Logtype B	**1kB**	approx. **250 million ($25*10^{7}$)**
Logtype C	**1kB**	approx. **50 million ($5*10^{7}$)**

EntryId is 16B

Usefulness window of log entries is 10 days

Need to keep in memory at most **250M** entries/day * **16** B/entry * **10** days = **40 GB**

Footprint: 2 machines per DC

Dimensioning EntryIdStore QPS

(Example using the B logtype)

Required throughput: **250 M** events/day / **86400** seconds/day ~= **2900 events/second**

For each joined entry, there will be at least 3 RPCs (read from Tailer, read from Joiner, write from Joiner)

5800 read QPS

2900 write QPS

Scaling EntryIdStore

Each Paxos round requires talking to all members.

Cross-continent latency is in the order of hundreds of ms (e.g., **150ms**) → each write RPC will take **~200ms**

Having only one in-flight commit round at a time would limit our throughput to **5-6 QPS**

Solutions: batching (cheap), sharding (less cheap).

To sustain 2900 write QPS, need 2900 / 6 ~= **480 queries/batch** (6 batches/s)

With 2 machines/DC, ~240 write QPS + 2900 read QPS (from RAM) per machine is reasonable.

Dimensioning Joiner

The bottleneck is QPS, most of the time spent waiting for I/O.

Average duration of a Join operation:

$$\underbrace{\text{EntryIdStore read}}_{\sim\mathbf{1ms}} + \underbrace{\text{EventStore read}}_{\sim\mathbf{10ms}} + \underbrace{\text{event creation}}_{\sim\mathbf{20ms}} + \underbrace{\text{EntryIdStore write}}_{\sim\mathbf{200ms}} + \underbrace{\text{write to (remote) disk}}_{\sim\mathbf{20ms}} = \sim\mathbf{250ms}$$

Average concurrent inbound connections: **2900** QPS * **0.25** s/query = **725**

Outbound connections: 4 * 2900 QPS ~= **11k**

Footprint: 1 machine

Global footprint

Common to B+C (for each DC):

- EventStore: 13 machines

For each log type (B or C):

- Tailer: 1 machine
- Joiner: 1 machine
- EntryIdStore replica: 2 machine

Total: (**13** machines/DC + (**4** machines/logtype/DC * **2** logtypes)) * **3** DCs = **21** machines/DC * **3** DCs = <u>62 machines</u>

Making the Design more Robust

Remove single points of failure (SPoF): no single-machine components.

- 2 machines each for Tailer and Joiner

Take into account traffic spikes. Assume the input distribution is not uniform but oscillates daily, with spikes up to 1.25x the average.

- Take 25% more machines

Final count of machines:

(**13** machines/DC + (**6** machines/logtype/DC * **2** logtypes)) * **3** DCs * 1.25 ~= 95 machines

Some additional notes

Reliability:

- each instance must be able to process **50%** of the streams
- individual machine failures: stateless and redundant components (e.g., by logtype pair A+B, A+C)
- catch-up capability is ~1.5x. Is this enough?

Observability:

- monitoring, alerting
- correctness verification

Other goodness: code reuse (e.g., tailing component, client-side library for EventIdStore)

Salim Virji

salim@google.com

Google™

References

The system described is a simplification of how **Photon**, Google's log-joining system, works.

For more information on Photon:

R. Ananthanarayanan et al. - *“Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams,”* in Proceedings of ACM SIGMOD’13 (http://research.google.com/pubs/pub41318.html)

For more information on Paxos at Google:

Chandra et al. - *“Paxos Made Live - An Engineering Perspective,”* in Proceedings of ACM PODC '07 (http://research.google.com/archive/paxos_made_live.html)