Site Reliability Engineering

Distributed PubSub

Non-Abstract Large System Design

NALSD

- “Non-Abstract Large System Design”
- Alternatively: SRE Classroom
- Large (“planet scale”) system design questions
- Hands-on workshops and exercises
- Non-abstract component:
  - Crunch numbers
  - Provision the system
- Resilient software systems
- Distributed architecture patterns

Google

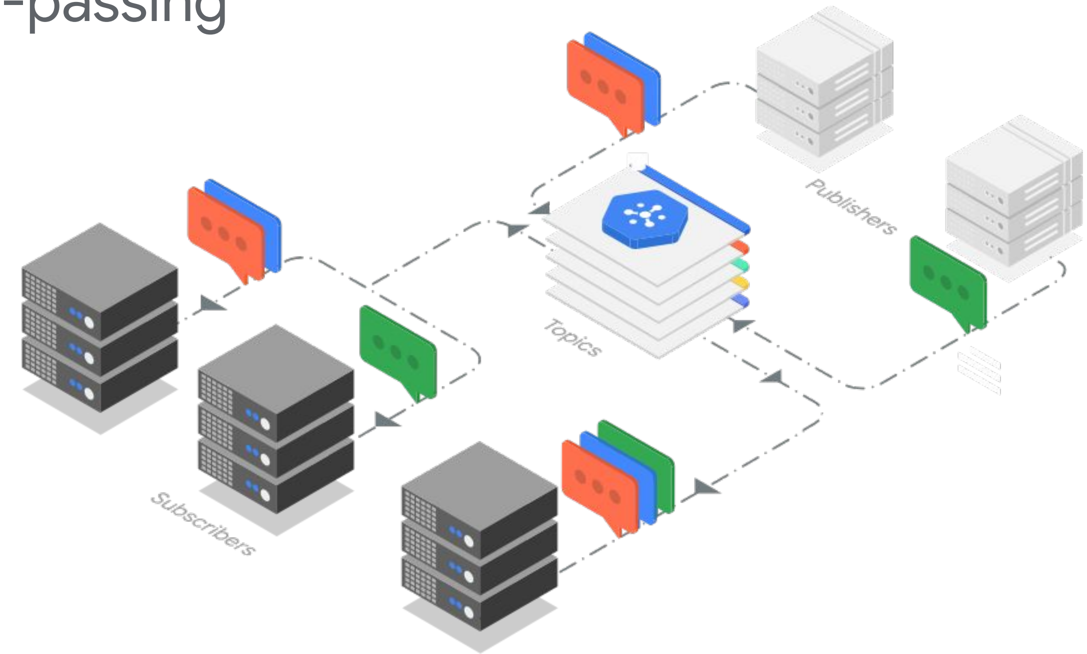
Agenda

- Introduction and problem statement
- “Let’s do it together”
- Breakout session 1: **Design for single datacenter**
- Single datacenter sample solution
- Breakout session 2: **Design for multiple datacenters**
- Multiple datacenters sample solution
- Breakout session 3: **Provision the system**
- Provision the system sample solution
- Wrap-up and conclusions

Introduction

Introduction: PubSub

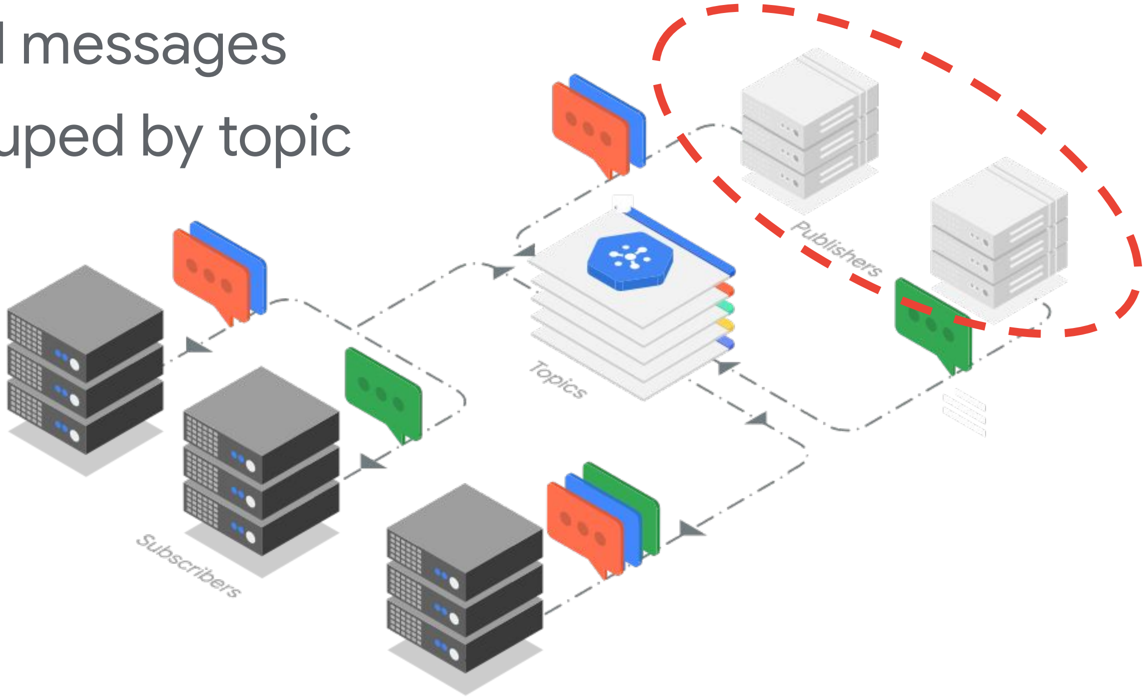
- Publish-Subscribe (PubSub)
- Asynchronous communication through message-passing



Google

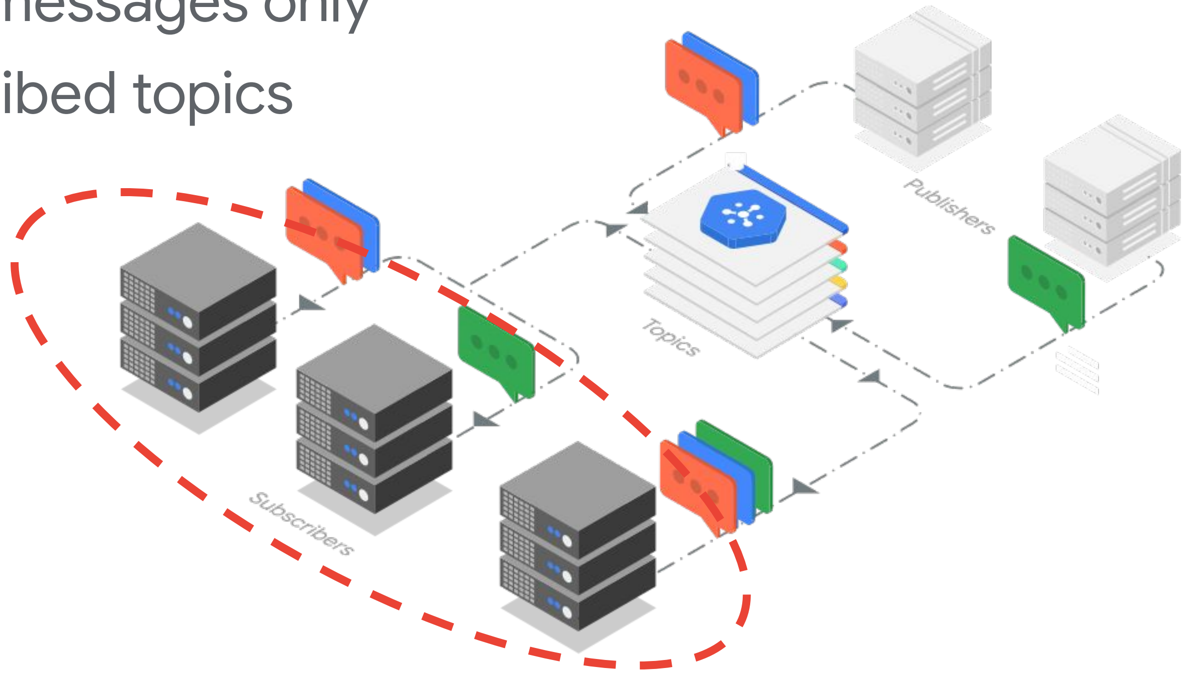
Introduction: PubSub

- Publishers: “producers” or “writers”
  - Senders of messages
  - Sends ordered messages
  - Messages grouped by topic



Introduction: PubSub

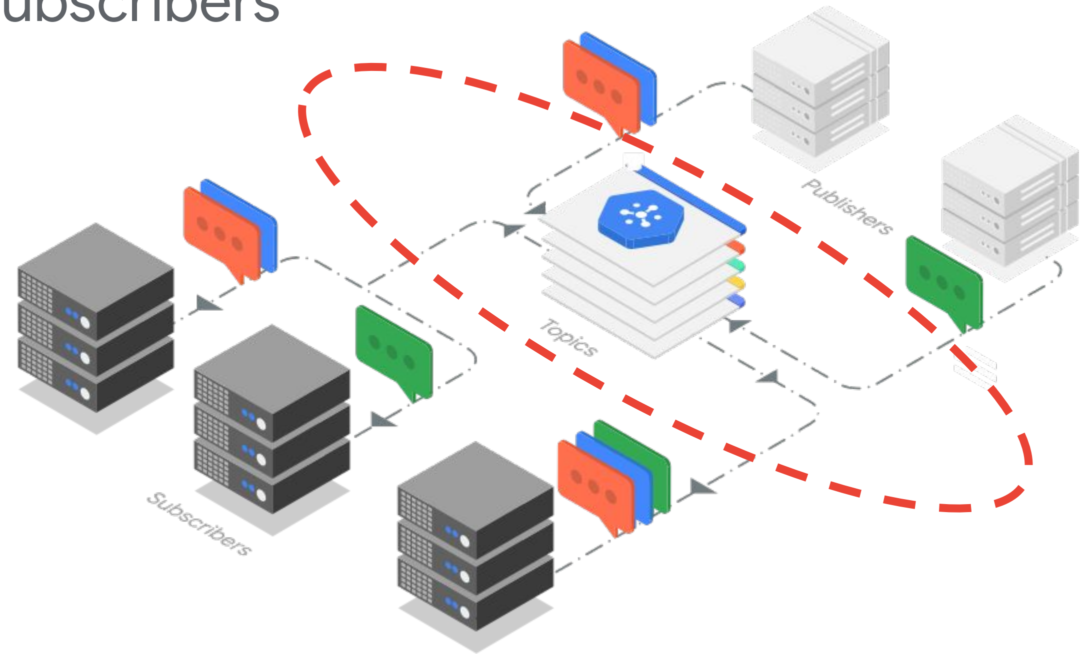
- Subscribers: “consumers” or “readers”
  - Subscribes to topics
  - Receives messages only for subscribed topics



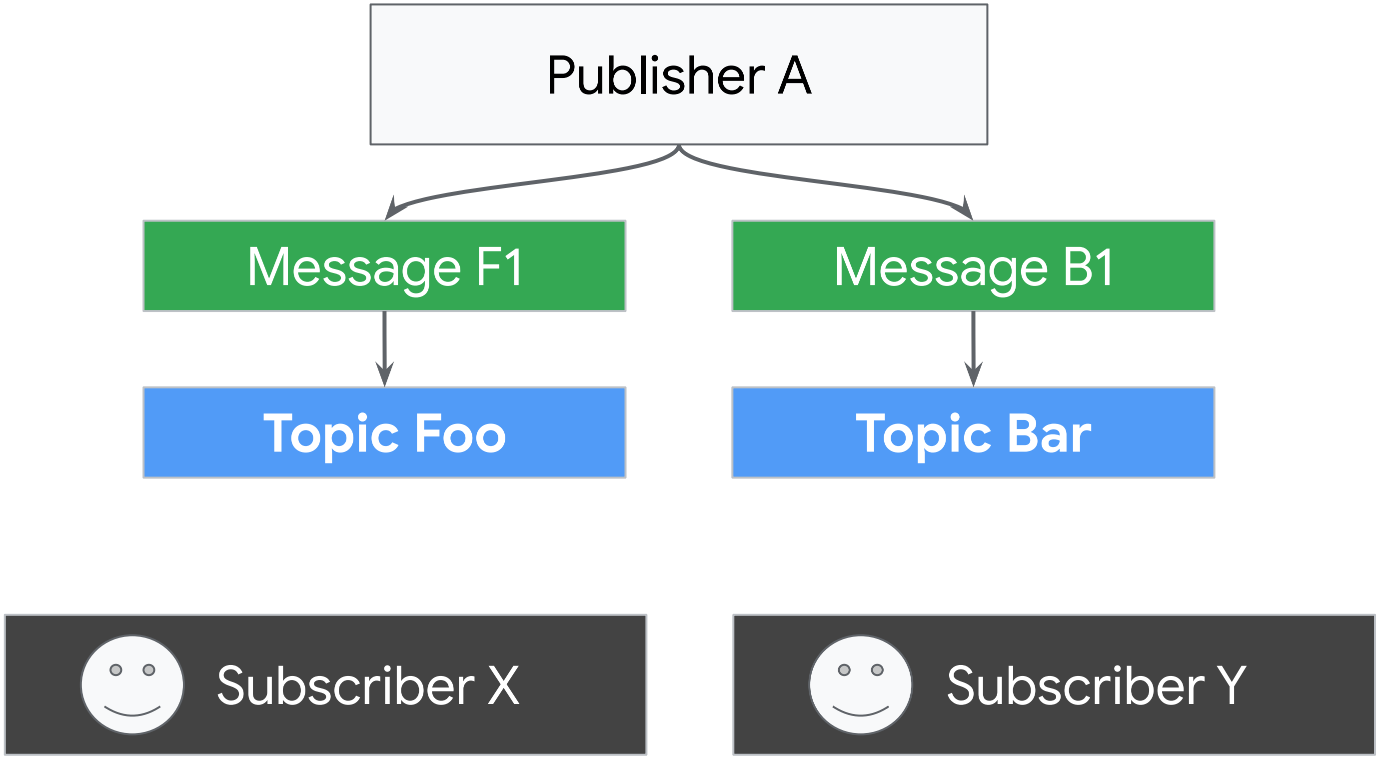
Google

Introduction: PubSub

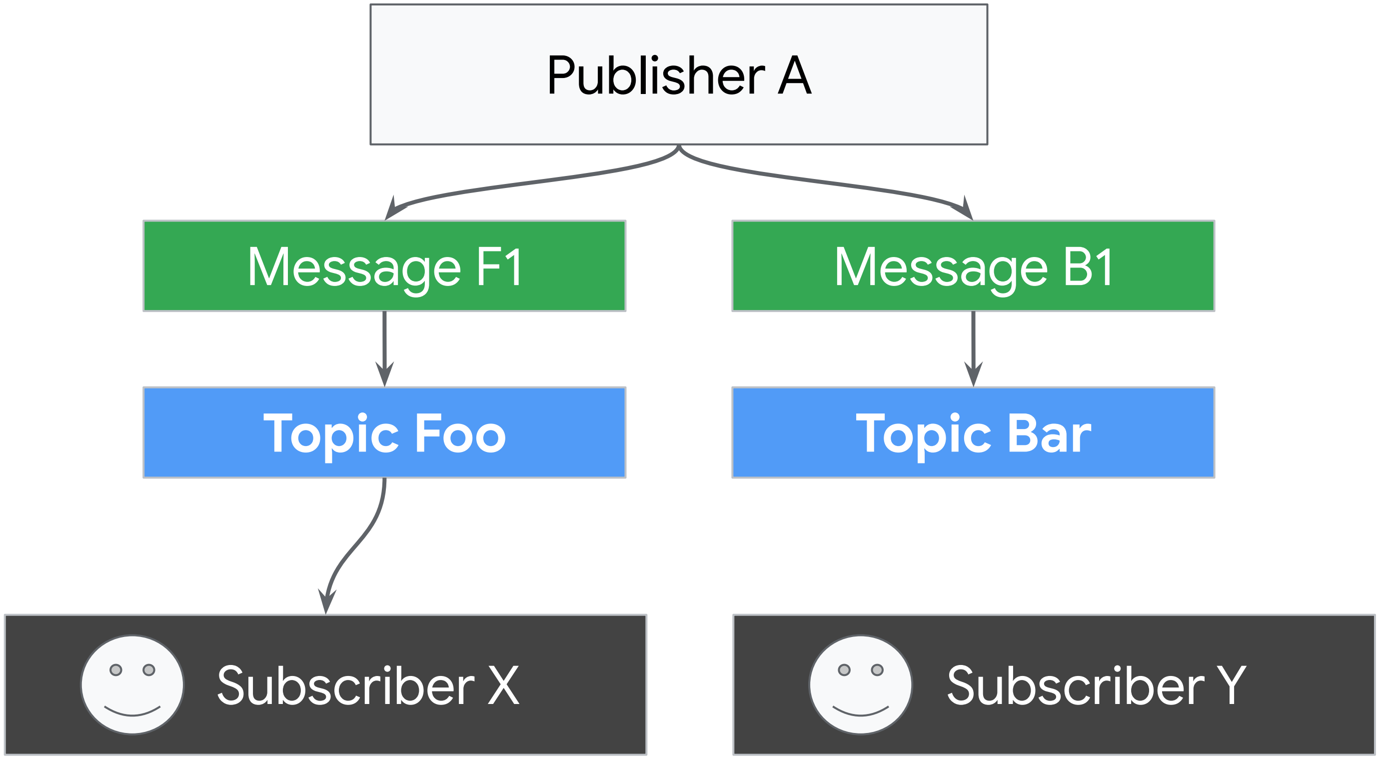
- Publishers **do not directly communicate** with Subscribers
- Subscribers **do not directly communicate** with Publishers
- Scale publishers/subscribers independently



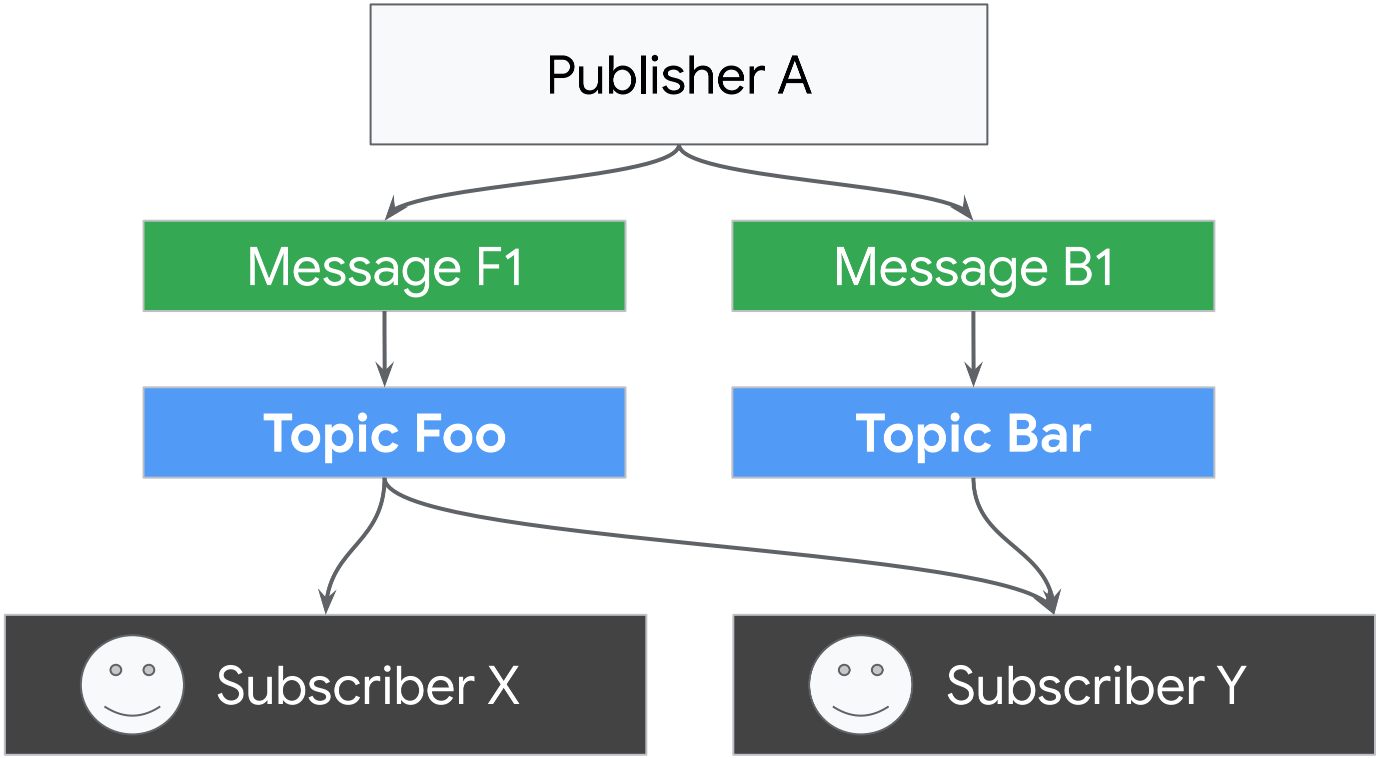
Introduction: PubSub



Introduction: PubSub



Introduction: PubSub



Problem Statement

Let's identify the problem at hand

"

Design a PubSub service that clients all over the world can use to read and write messages.

Gather Requirements

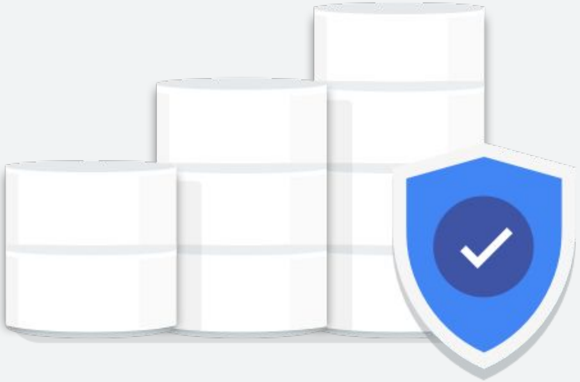
Let's identify what we know and what we need

Requirements

Background

What we have:

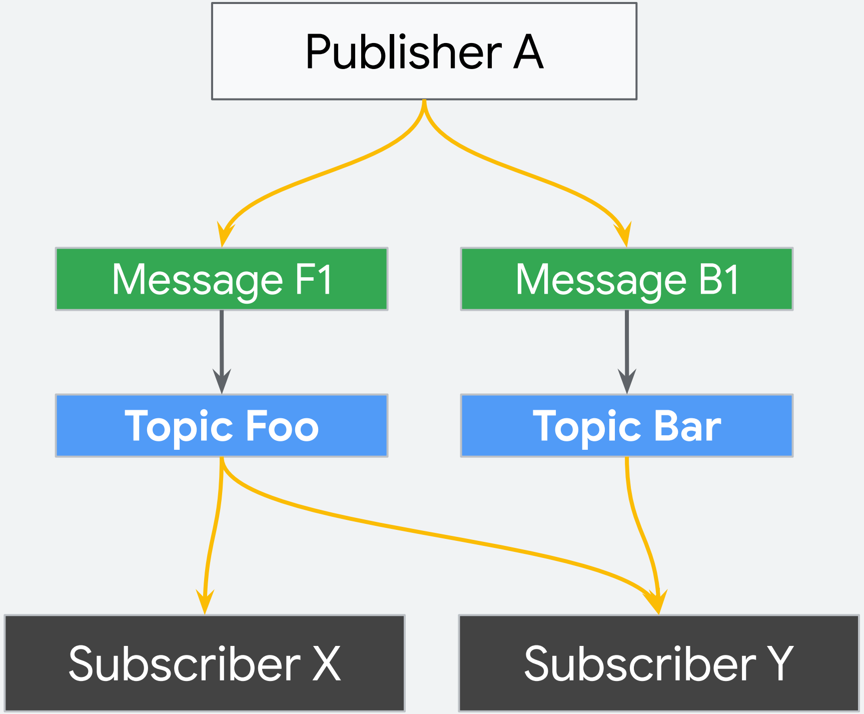
- Three datacenters (DCs):
  - New York
  - Seattle
  - Kansas City
- Reliable storage system
  - Distributed!
- Reliable network
- Authentication & Authorization



Requirements

What we need:

- A way to publish messages
    - Ordered
    - Grouped by topic
- A way to receive messages
    - Ordered
    - Grouped by topic
- Message persistence



Requirements

- Each DC runs the PubSub service we are designing
- Clients all over the world read and write messages
- Large volume of messages per day
- Uneven distribution of traffic over time

Requirements - What Does PubSub Do?

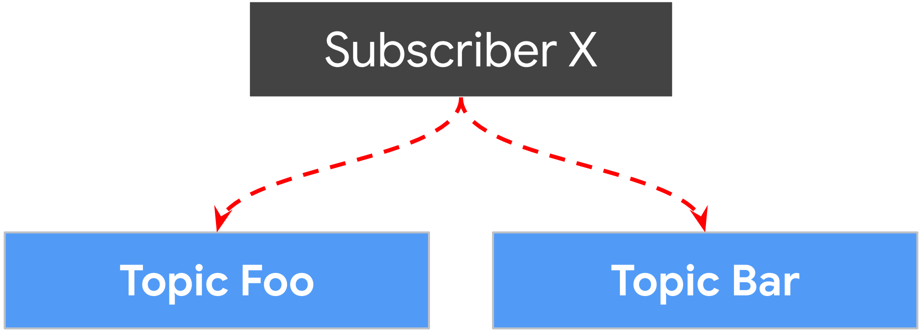
- Communicate **ordered** messages, **grouped** by topic
- Readers/writers can connect to any DC
- Users expect the same level of service from all DCs
- If a DC goes down, the user will automatically get connected to another one (this is already provided as a service)
- Once a DC recovers, it goes back to full service

Requirements - PubSub API

- Topics are identified by their **`topic_id`**.
- Readers are identified by their **`consumer_id`**.
- Readers will explicitly subscribe to topics.

- **`Subscribe(topic_id, consumer_id):`**

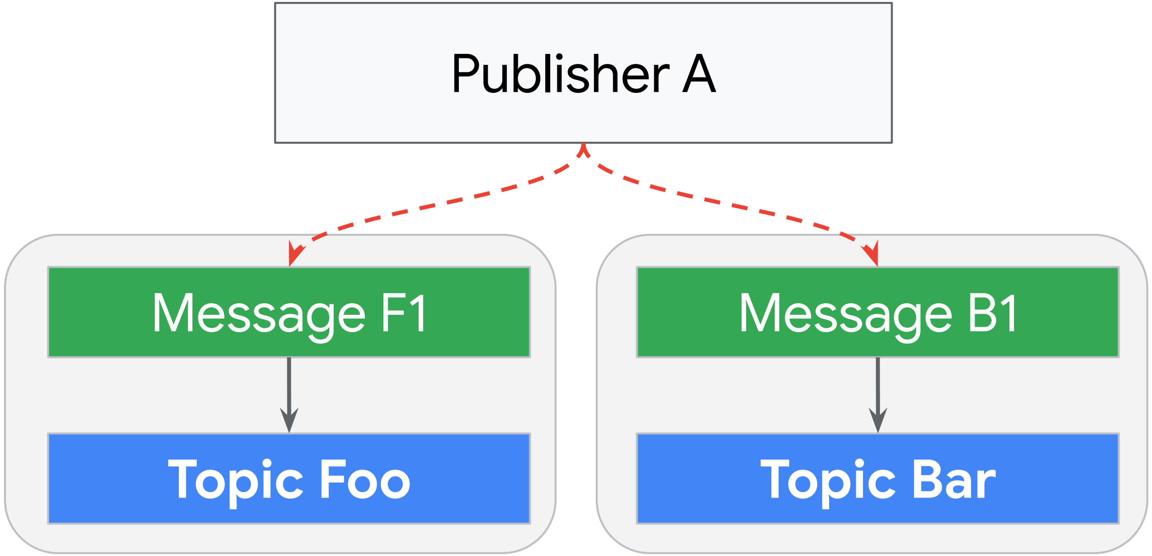
  Subscribes the given consumer to the given topic.



Requirements - PubSub API

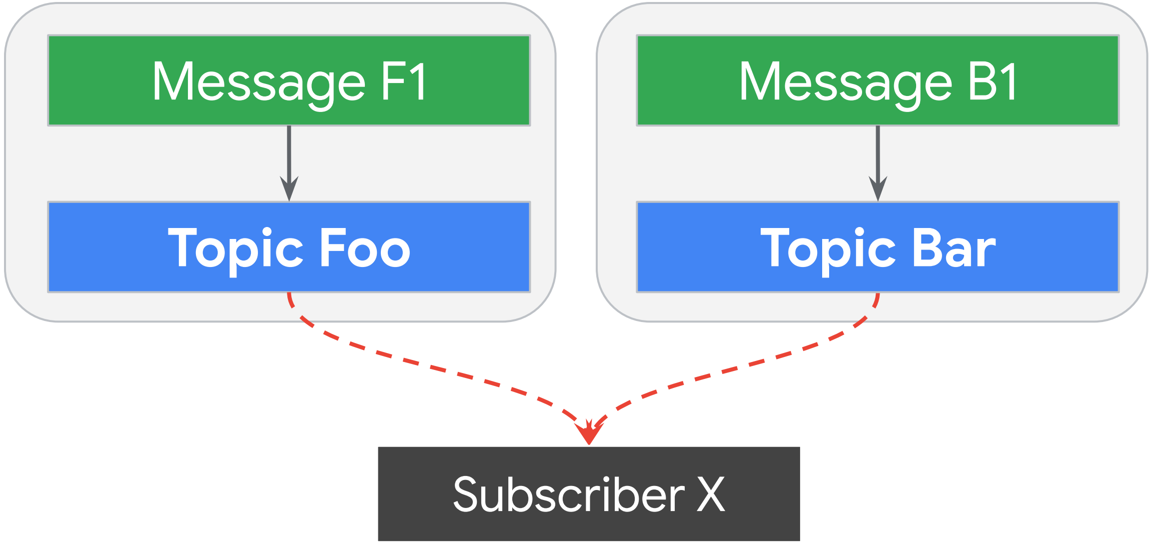
- **`Push(topic_id, message):`**

  Append the message to the given topic.

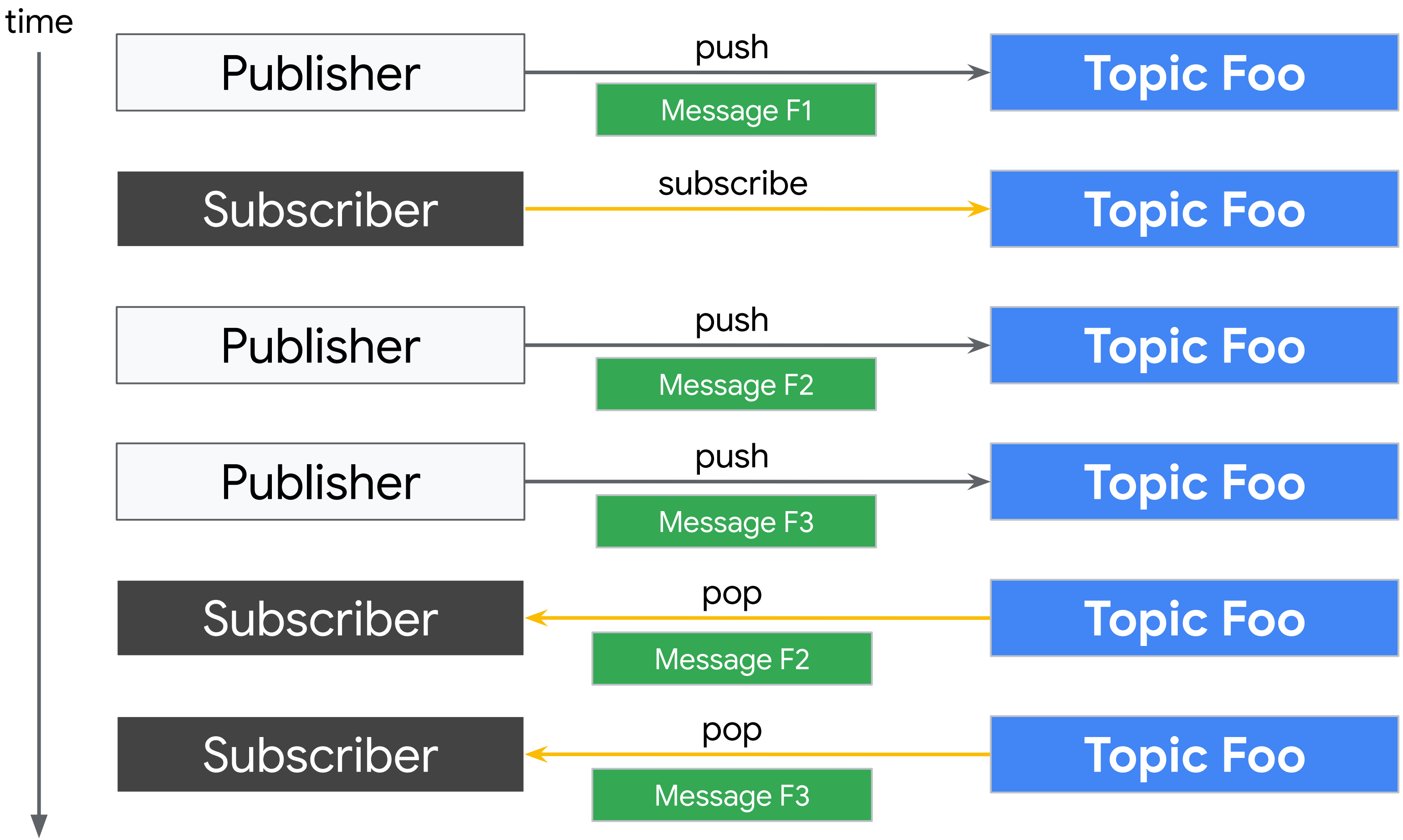


Requirements - PubSub API

- **`Pop(topic_id, consumer_id):`**
  Read the next message (in order) for the given topic.



Requirements - PubSub API

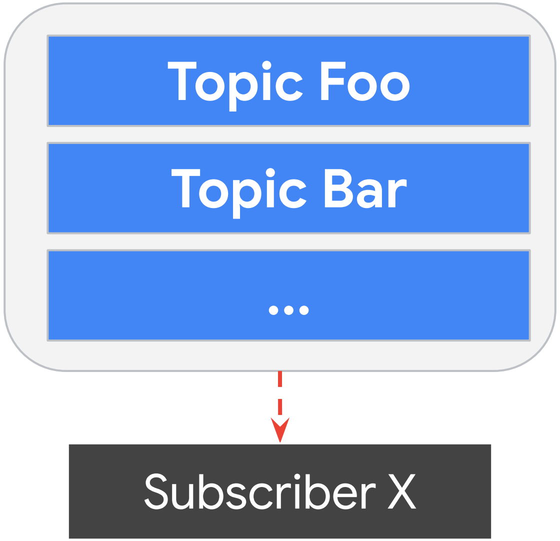


Google

Requirements - PubSub API

- **`List():`**

  Returns a list of all available topics.
- Not in scope for this exercise.



Google

Service Level Terminology

- **SLI: service level <u>indicator</u>**

  A quantifiable (numeric) measure of service reliability.

- **SLO: service level <u>objective</u>**

  A reliability target for an SLI.

- **SLA: service level <u>agreement</u>**

  SLO + consequences when SLO is violated

Requirements - SLO

Availability

- PubSub must continue working under peak load even if one datacenter goes down

Latency

- 99% of API calls must complete within 500ms
- 99% of pushed messages must be available for pop anywhere in the world within 1s

Requirements - SLO

Correctness

- At-Least-Once delivery
- 100 day message retention
- System can lose 0.01% of enqueued message per year

Further details, including volumes of data, are in the workbook handouts.

Let’s do it together: push()

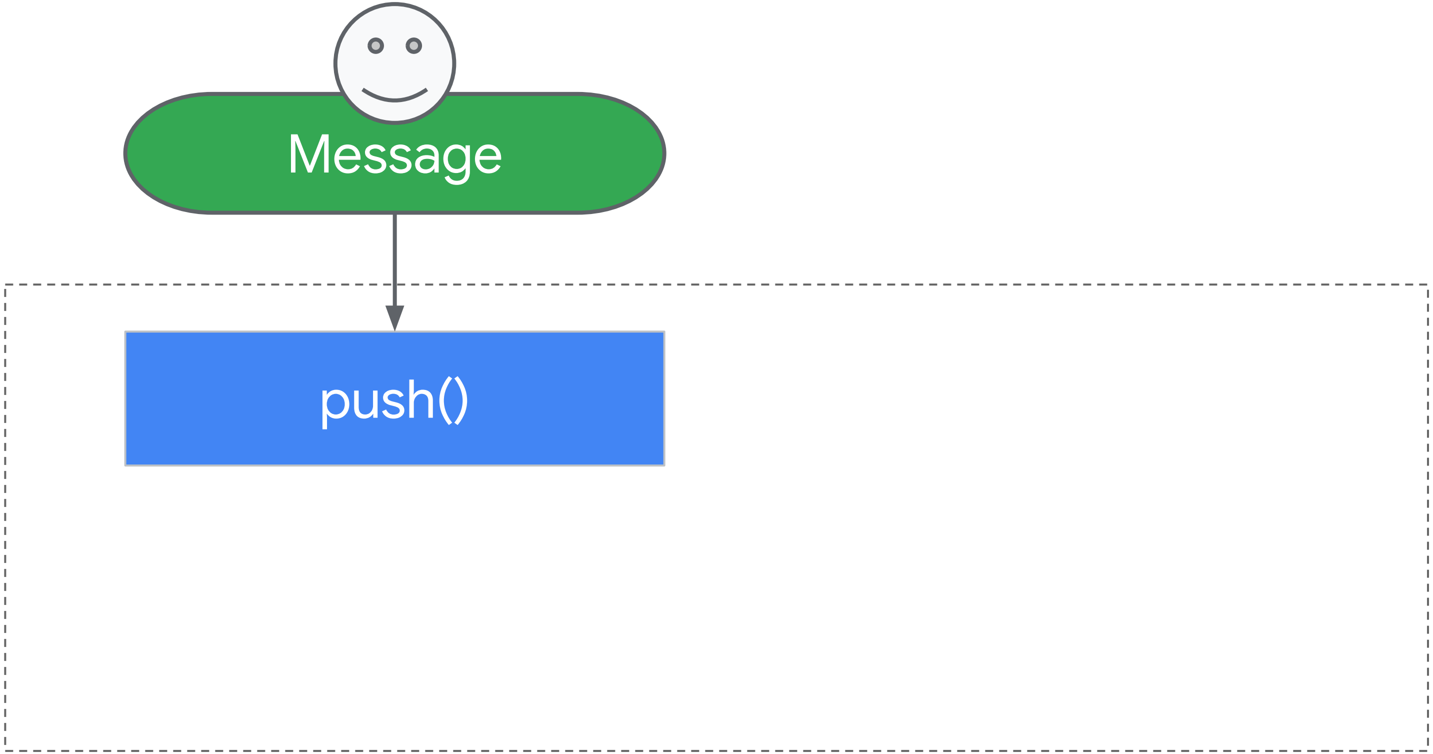
Requirements Recap

- Global PubSub Service
- Three datacenters (DCs):
  - New York
  - Seattle
  - Kansas City
- Clients all over the world write (**`push`**) and read (**`pop`**)
- Large volume of messages per day
- Uneven distribution of traffic over time

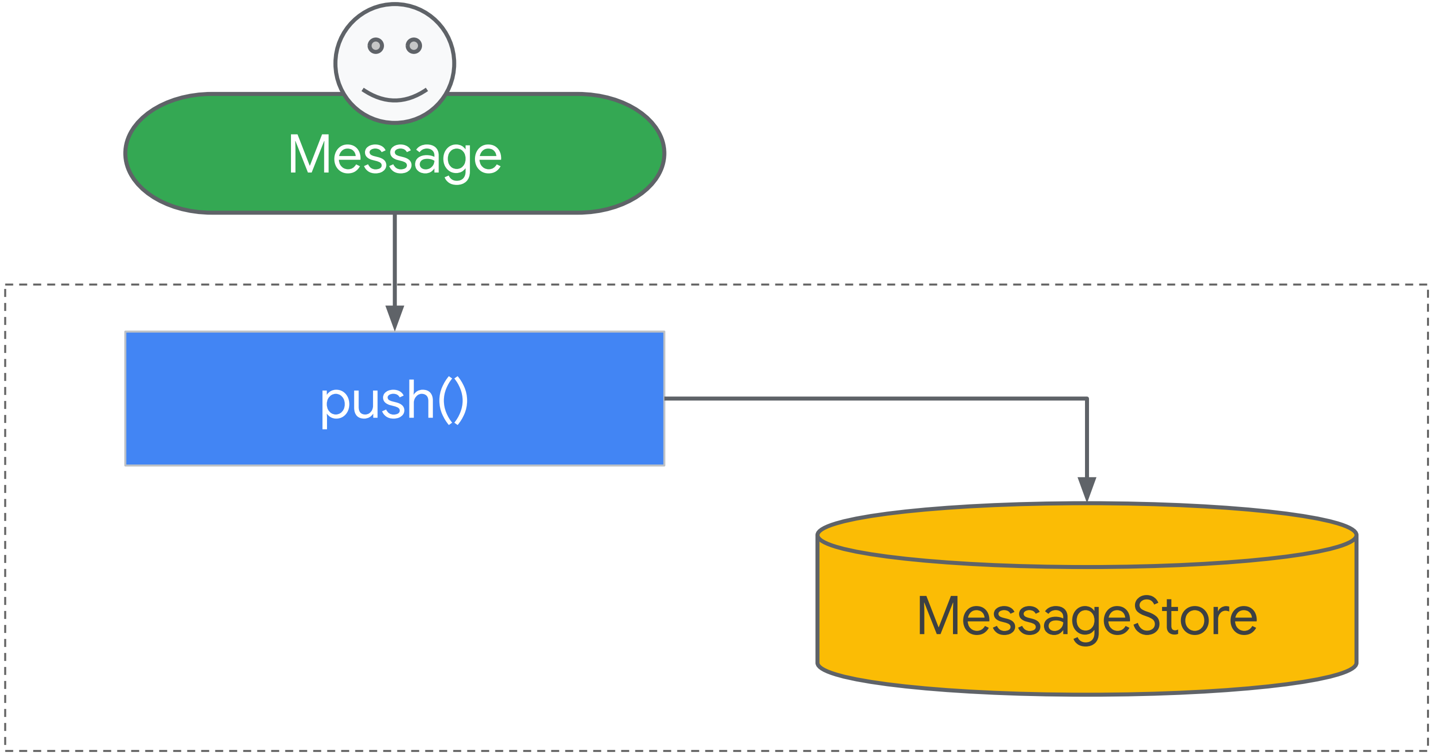
push()

Let's design the API call that receives messages.

Pushing a message

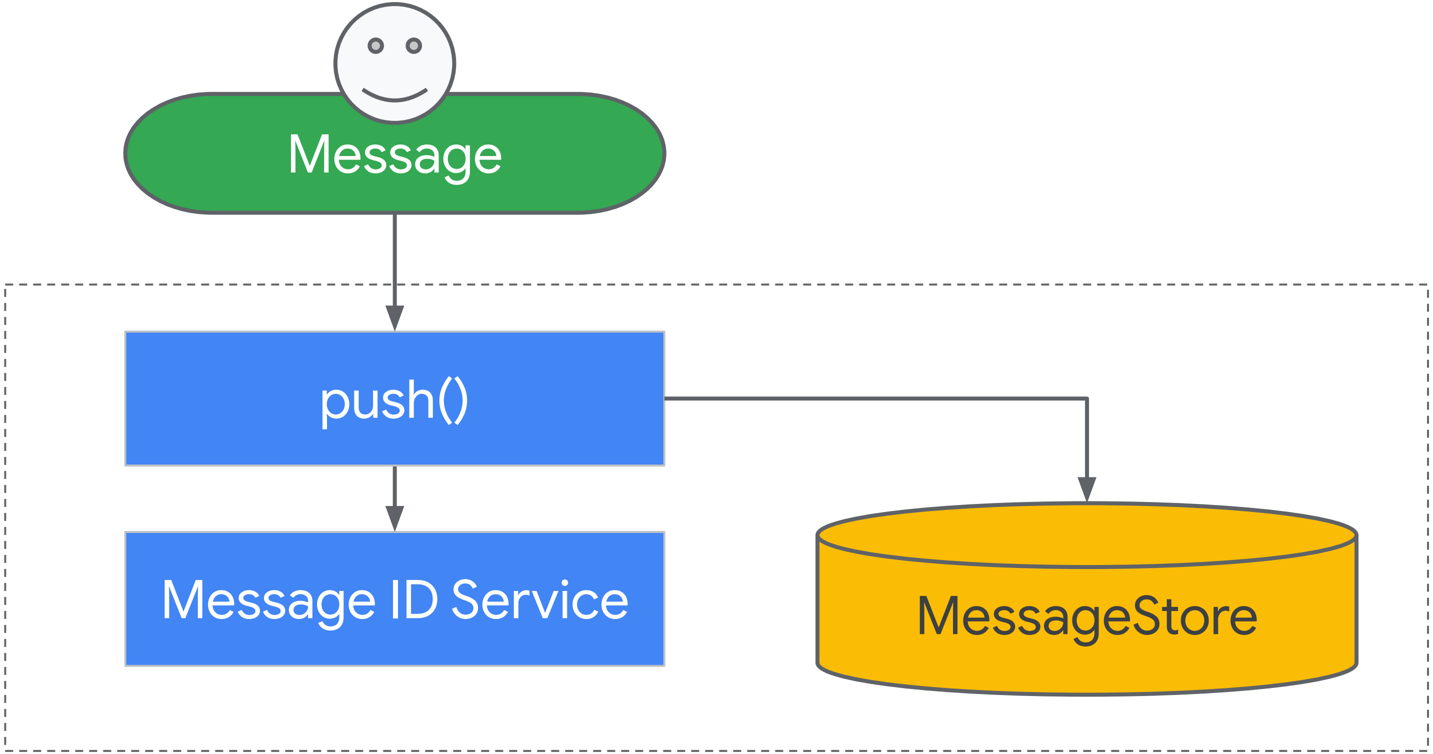


Start by storing the messages...



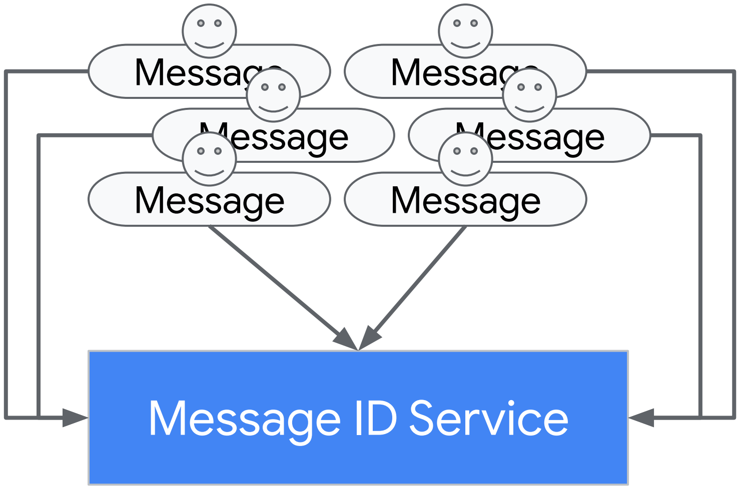
Google

Assign message IDs for storage...



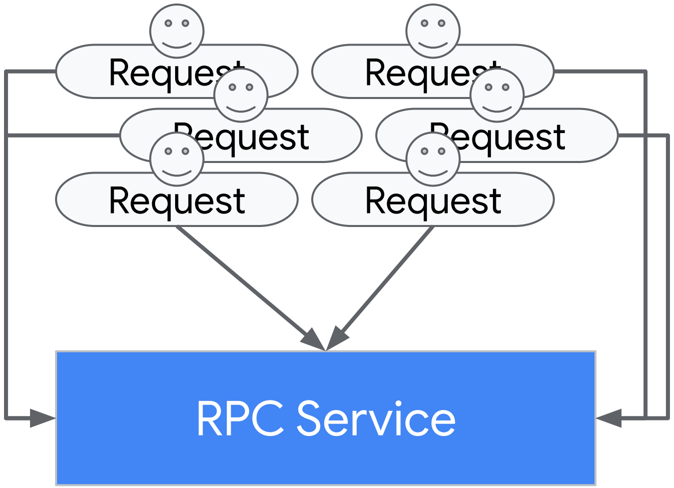
More on the Message ID Service

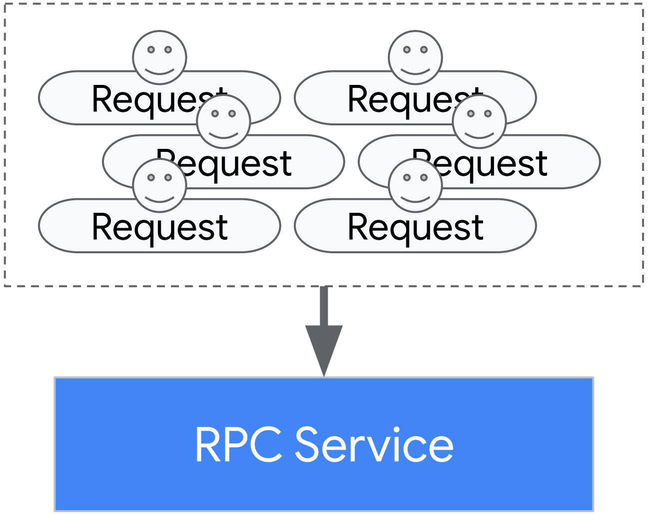
- Assign **unique** IDs for message within a topic
- Assign **ordered** message IDs for simple ordered lookup



Batch Operations

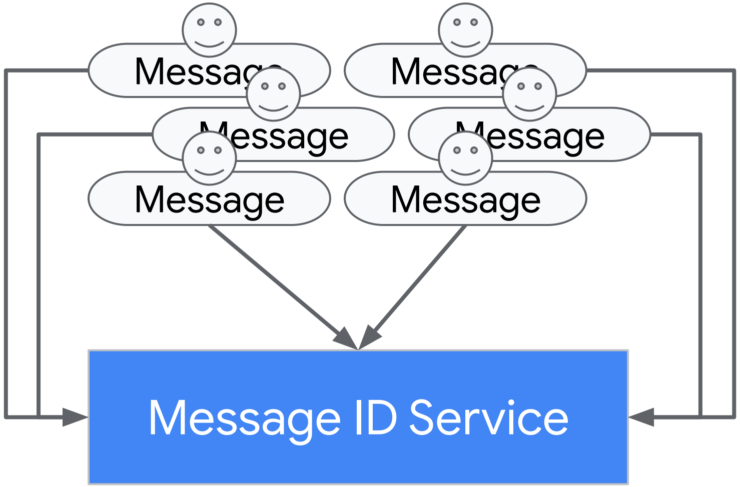
- Address **bandwidth** or **throughput** bottlenecks
- May be supported alongside singular operations
- Basically: stuff multiple requests into a single RPC

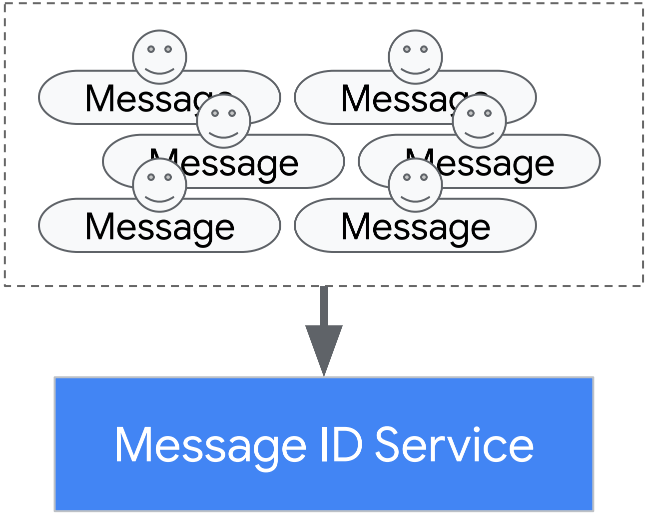




More on the Message ID Service

- Assign **unique** IDs for message within a topic
- Assign **ordered** message IDs for simple ordered lookup
- Performance optimizations: batch operations

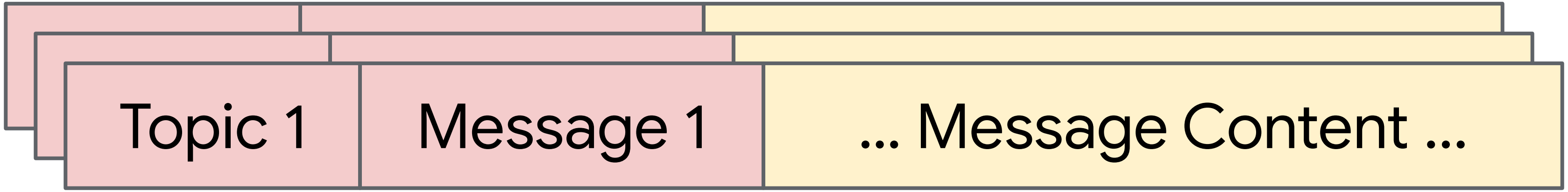




Google

More on the MessageStore

Key: **Topic ID, Message ID**
Value: **Message Content**



More on the MessageStore

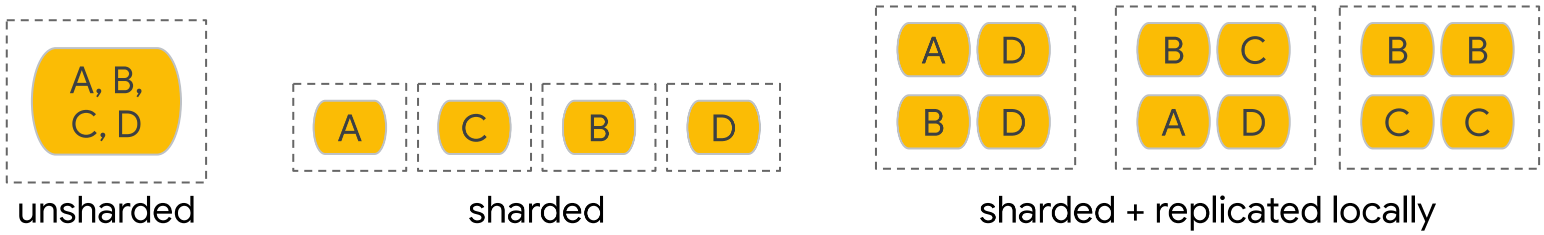
- Distributed file system
  - Storage abstractions
  - write(), read(), implemented already
  - Supports configurable replication strategy

Message Store Sharding

- Need to retain 100 days worth of messages
- 100 days * ... = 25TB of data → too big for one machine :(

Sharding

- Address **storage size** bottlenecks
- Basically: split your data into multiple buckets, and store those buckets separately, possibly multiple copies of each bucket
- Sharding mechanism should be flexible
- Consistency and fault tolerance
- A single disk failure should not cause data loss
- Consider replicating shards locally (local reads are cheapest)



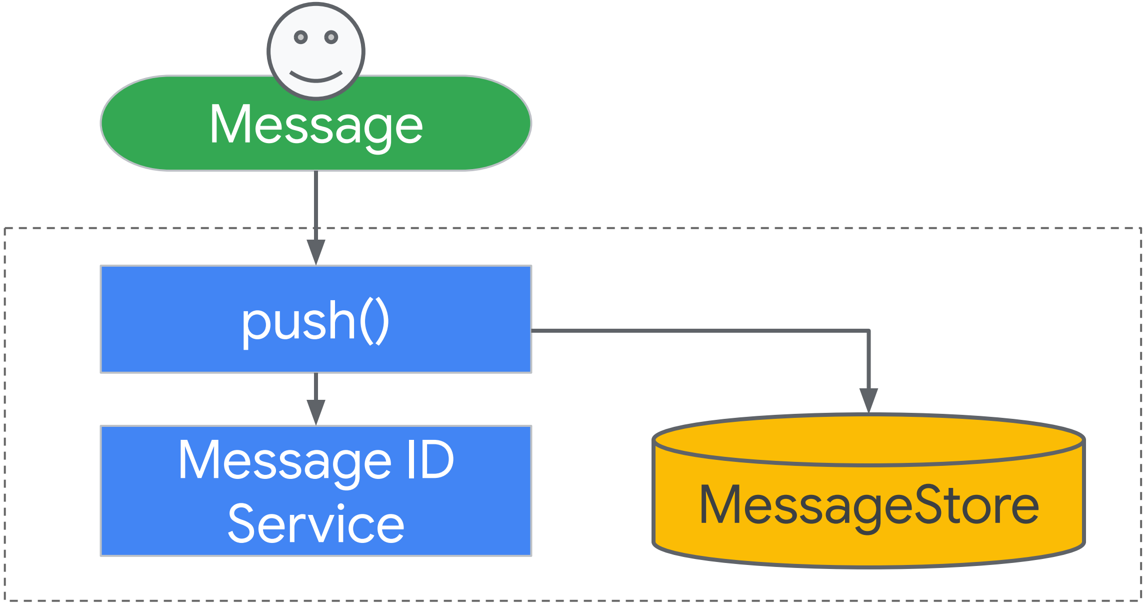
Google

Message Store Sharding

- Need to retain 100 days worth of messages
- 100 days * ... = 25TB of data ⟶ too big for one machine :(
- **Sharding to the rescue!**
- Keep multiple copies (replicas) of each shard:
  - Greater resilience
  - ... and performance too (local reads are cheap)!

Flow overview: push()

1. Get message ID from Message ID Service
2. Write message to MessageStore
3. Ack receipt of message



Reminder: don’t sweat it!

- Designs will be different, with different abstractions: that’s okay!
- Focus on the process of designing something end-to-end
- Think about high level concepts, rather than nitty details
- Think about trade-offs of different design decisions
- Make assumptions explicit
- Call out risks
- Simplify the problem
- If working in a group, discuss ideas and use each other as resources!

Rules of engagement

- Assume good intent
- Respect each other
- Speak up and share information
- Let everybody speak
- Ask questions

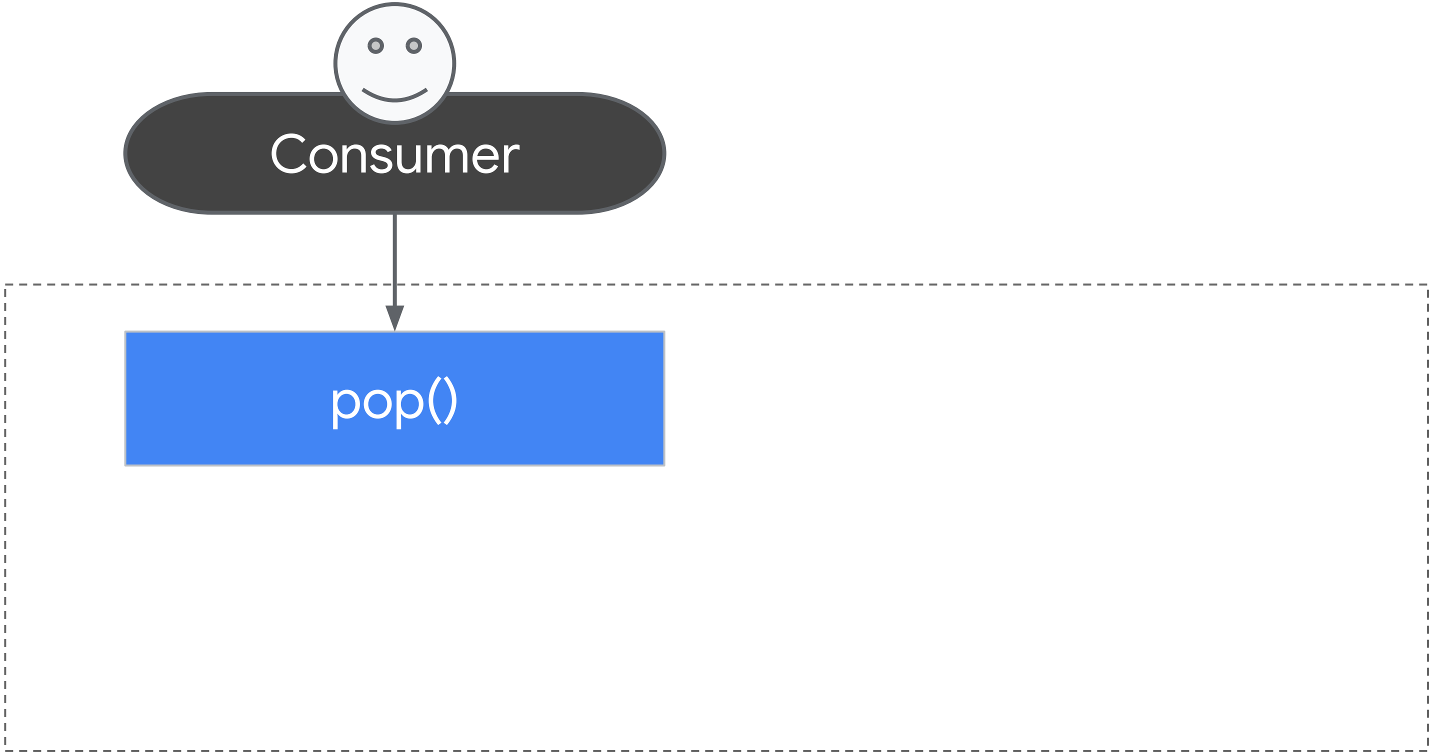
Most importantly, have fun!

Breakout Session 1:
Single Datacenter (40 minutes)

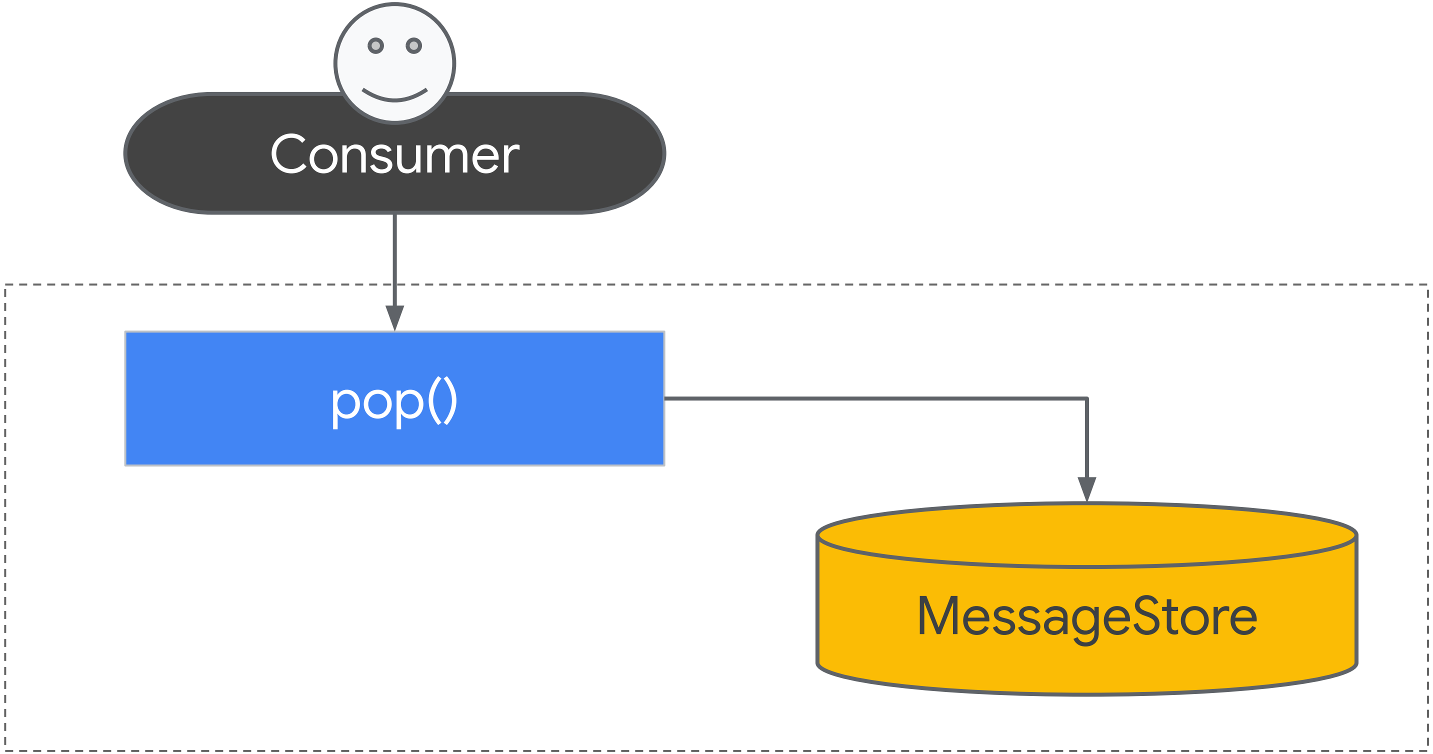
Goal:
Design a working system that fits in a single datacenter.

Break: 5 Minutes

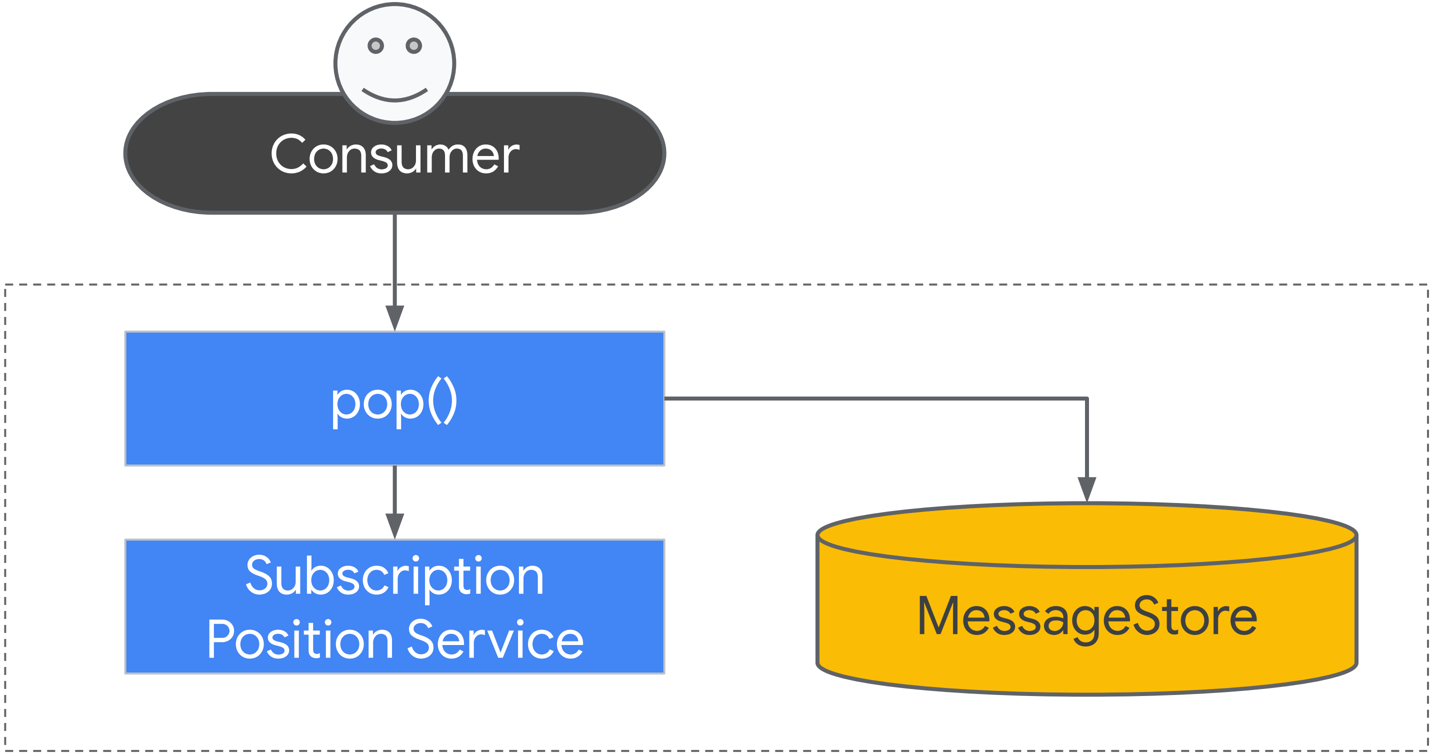
Reading a message



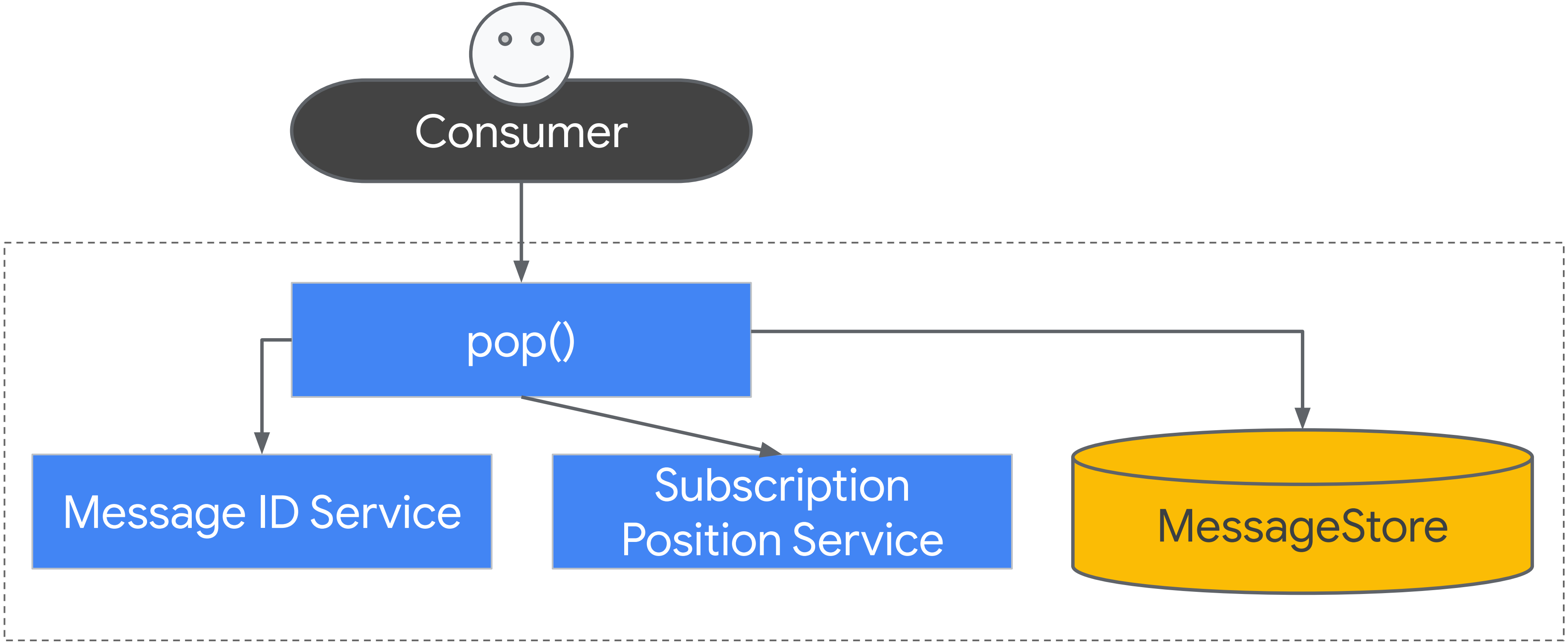
Reading a message



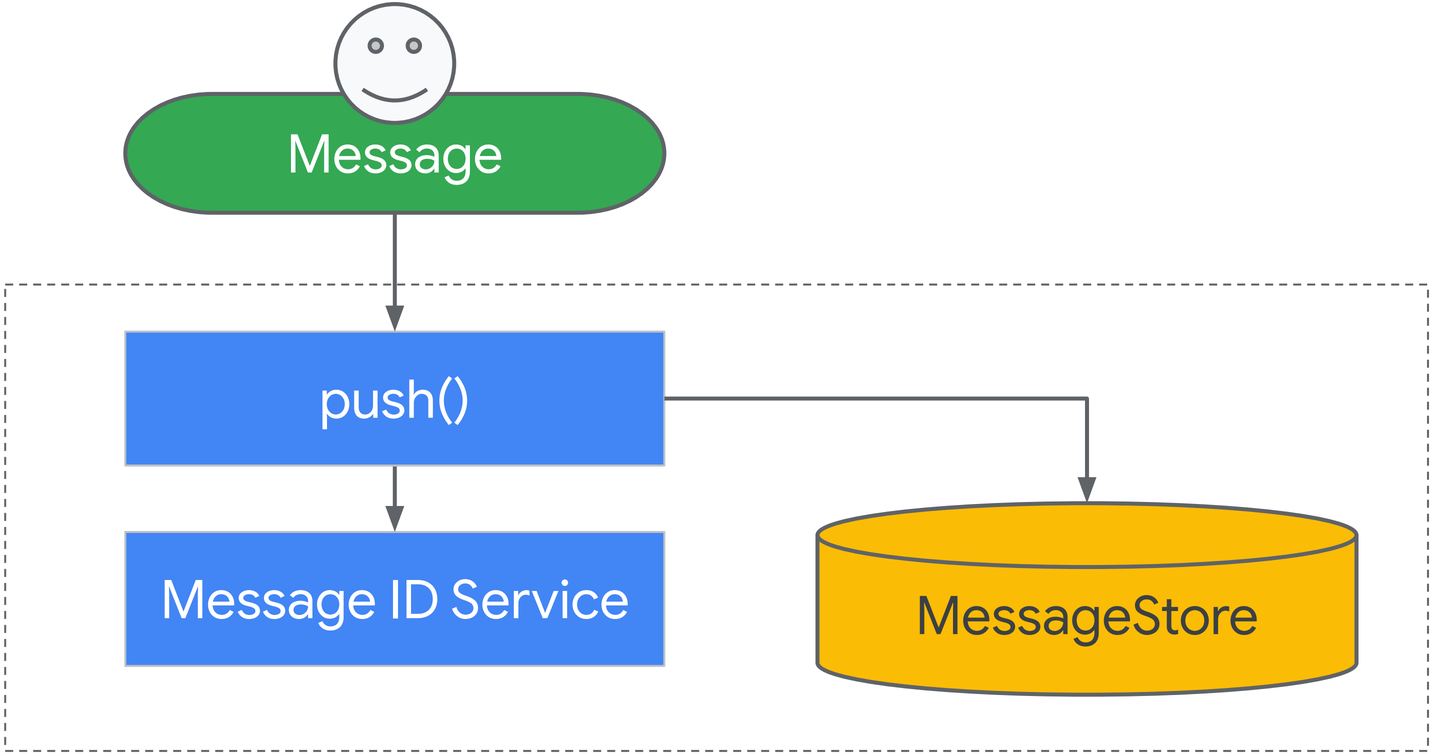
Reading: getting the "next" message



Next, read the messages on demand...



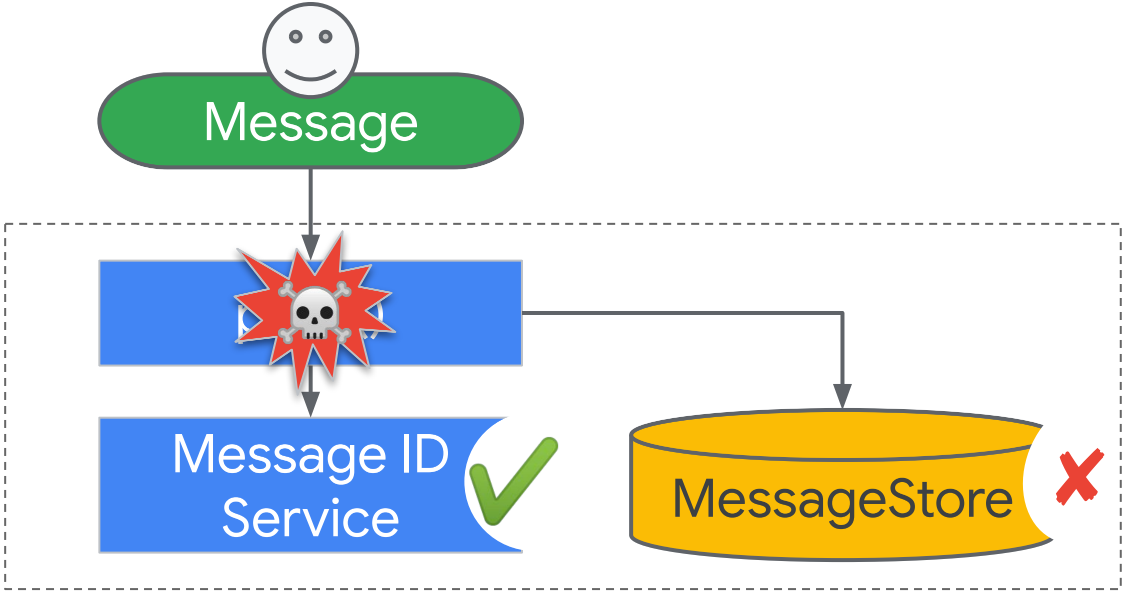
Reminder of how push() works...



Google

Error Handling: pop()

- Message IDs are consecutive... almost.
- Gaps can arise if push() service crashes after allocating ID, but before message is successfully written to storage.



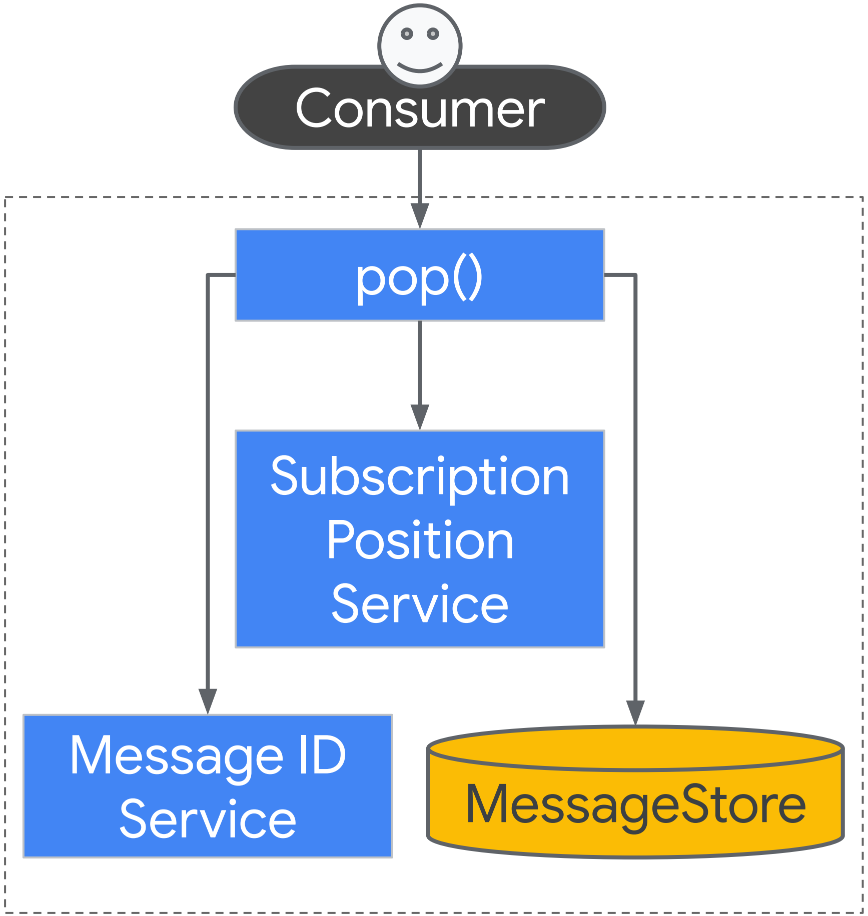
Google

Error Handling: pop()

- Detect error upon read
- Increment ID and keep reading until the next message is found
- Do not read past the end of the topic
- Some latency impact; expect to be rare
- Performance optimizations:
  - Batch reads
  - Readahead cache
  - Bloom filter on storage service

Flow Overview: pop()

1. Get latest written message ID from Message ID Service
2. Get latest read message ID from Subscription Position Service
3. Increment the read message ID
4. If at the end of topic, return
5. Read message from storage
6. Return the message to consumer
7. Update subscription position for consumer and topic

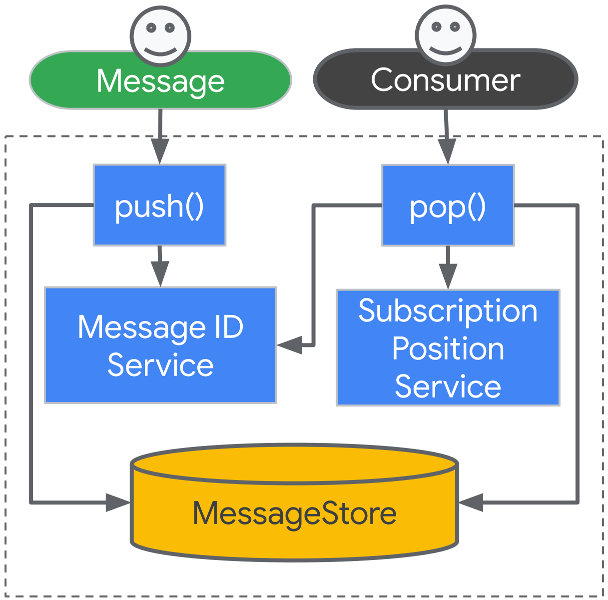


Breakout Session 2:
Multiple Datacenters (30 minutes)

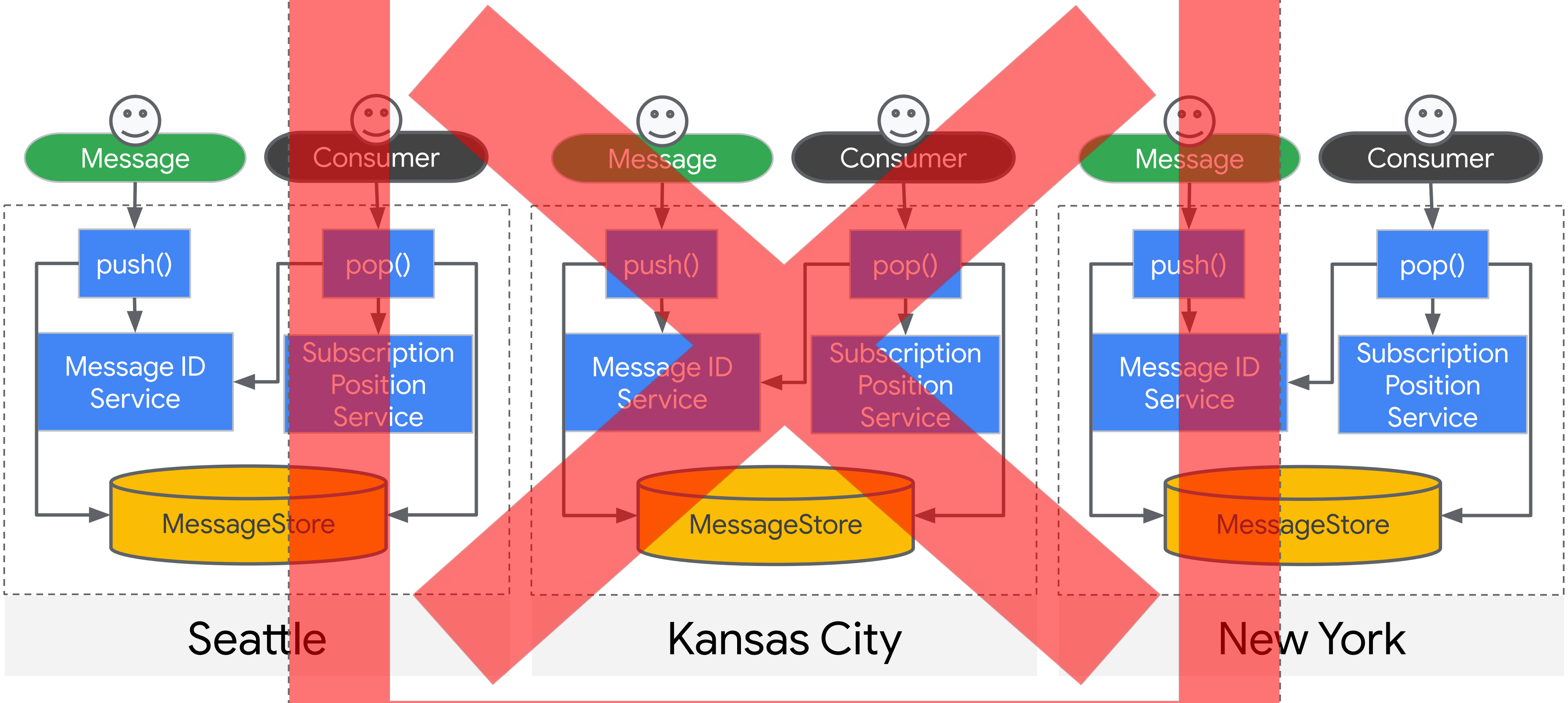
Goal:
Extend the design to work correctly in multiple datacenters.

Break: 5 Minutes

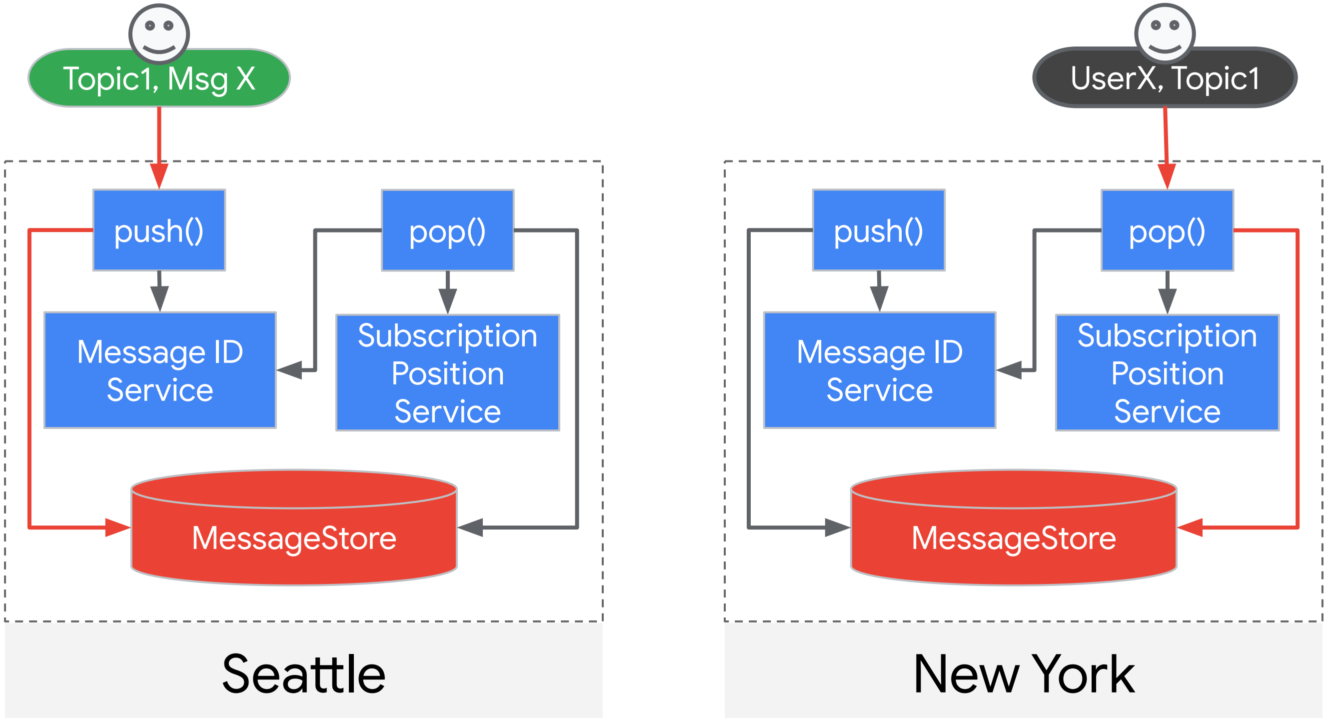
Single Datacenter Design



One for each datacenter...?



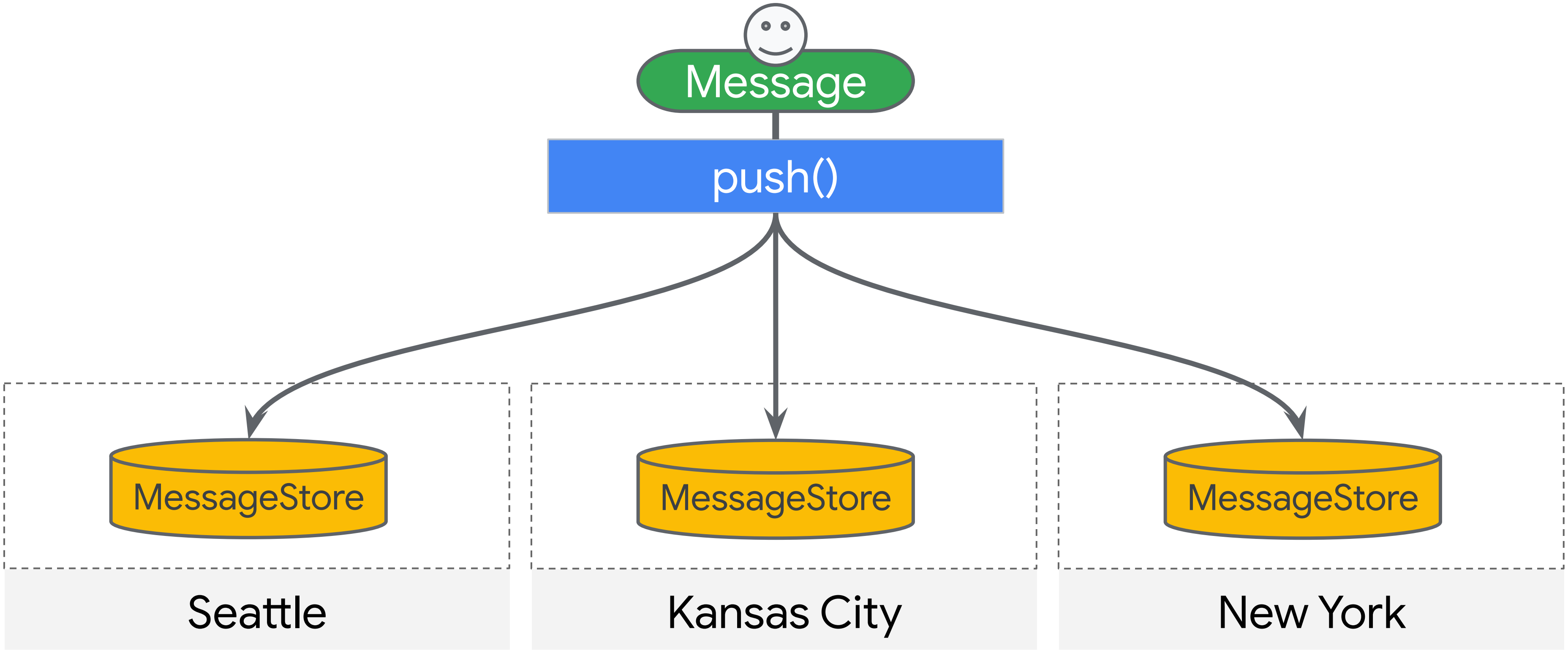
Partitioned MessageStore



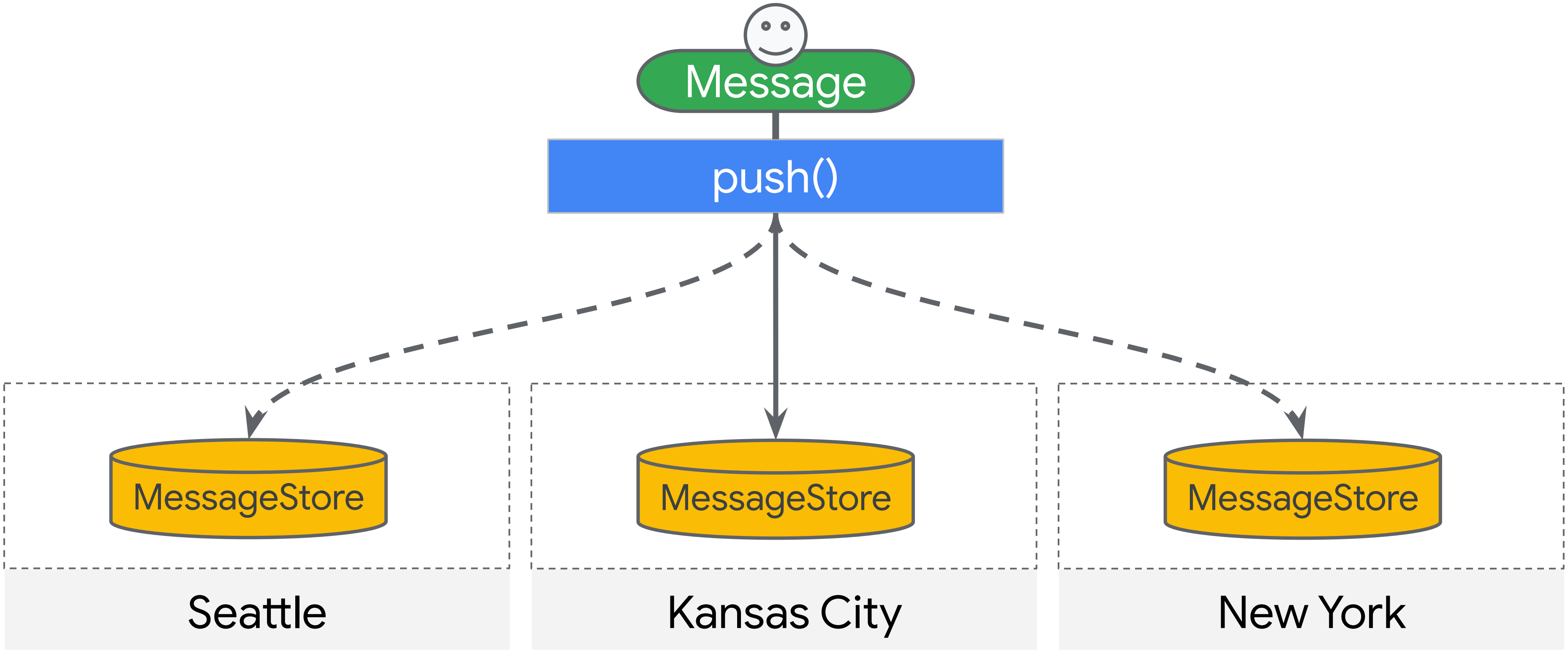
MessageStore Replication

- Pushes can arrive at any datacenter
- Need to be able to pop messages from any datacenter, even at a different datacenter than where it arrived
- Need to replicate messages to every datacenter
- Factors to consider:
  - Consistency
  - Fault tolerance
  - Availability

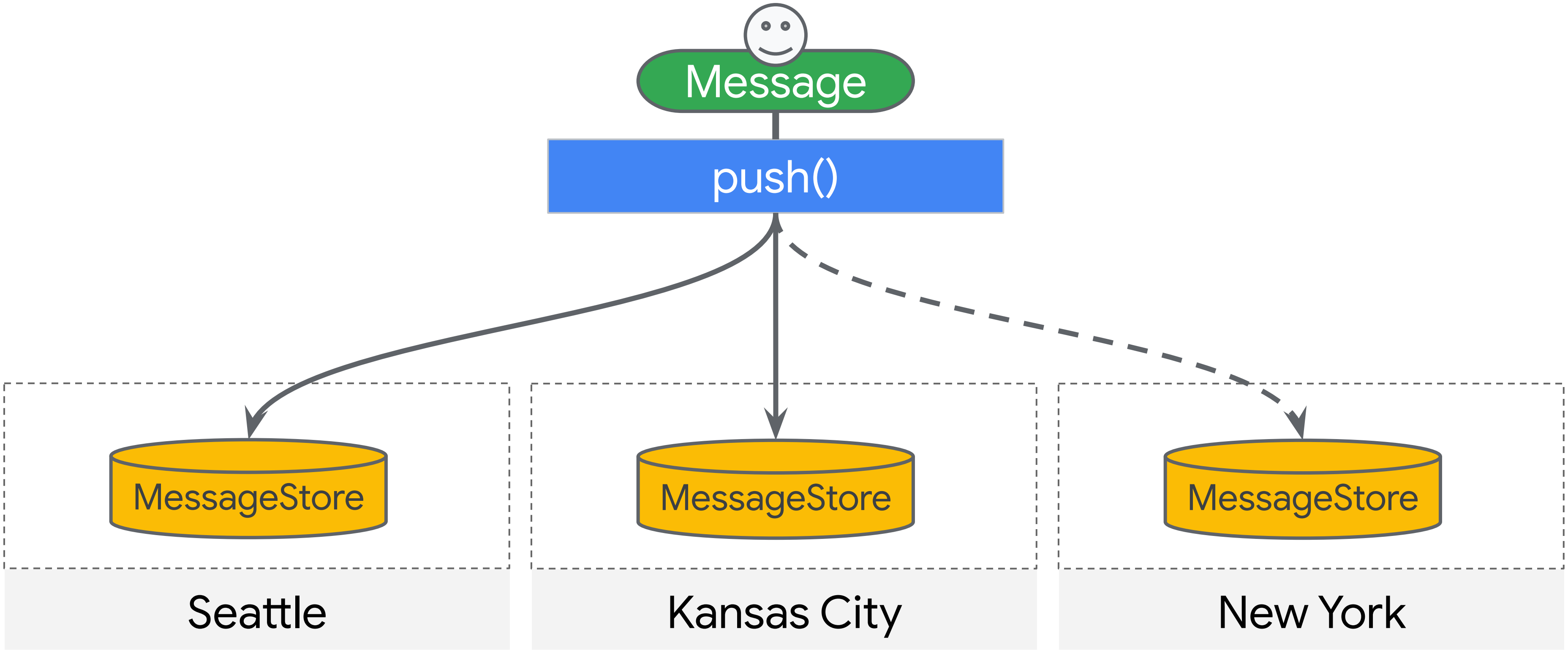
Replication: synchronous



Replication: asynchronous



Replication: hybrid



Google

MessageStore Replication: Tradeoffs

	Push Latency	Pop Latency	Data Durability
Synchronous Replication	High	Low	High
Asynchronous Replication	Low	High	Low
Hybrid Replication	Medium	Medium	Medium

Google

MessageStore Replication

- Asynchronous writes: ~10ms response time
- Can we afford the data loss?
- Reminder:
  - Can lose 0.01% of pushed messages per year
  - 99% of messages must be available for pop from any location in 1 second or less

5,000 topics * 10,000 msg / day / topic = 50M msg / day
⟶ Can lose 5k messages per day.

Async Replication

90k sec/day * 1 msg/sec/thread

= 90k msg / day / thread

parallelize processing to handle the entire load...

(50M msg / day) /

(90k msg / thread) =

~600 threads / day

(i.e. **concurrent** loads / day)

Reminders:

- 50M msg / day
- 99% of messages must be available for pop from any location in 1 second or less
- ~90k seconds / day
- Assume 1 second replication delay

Google

Async Replication

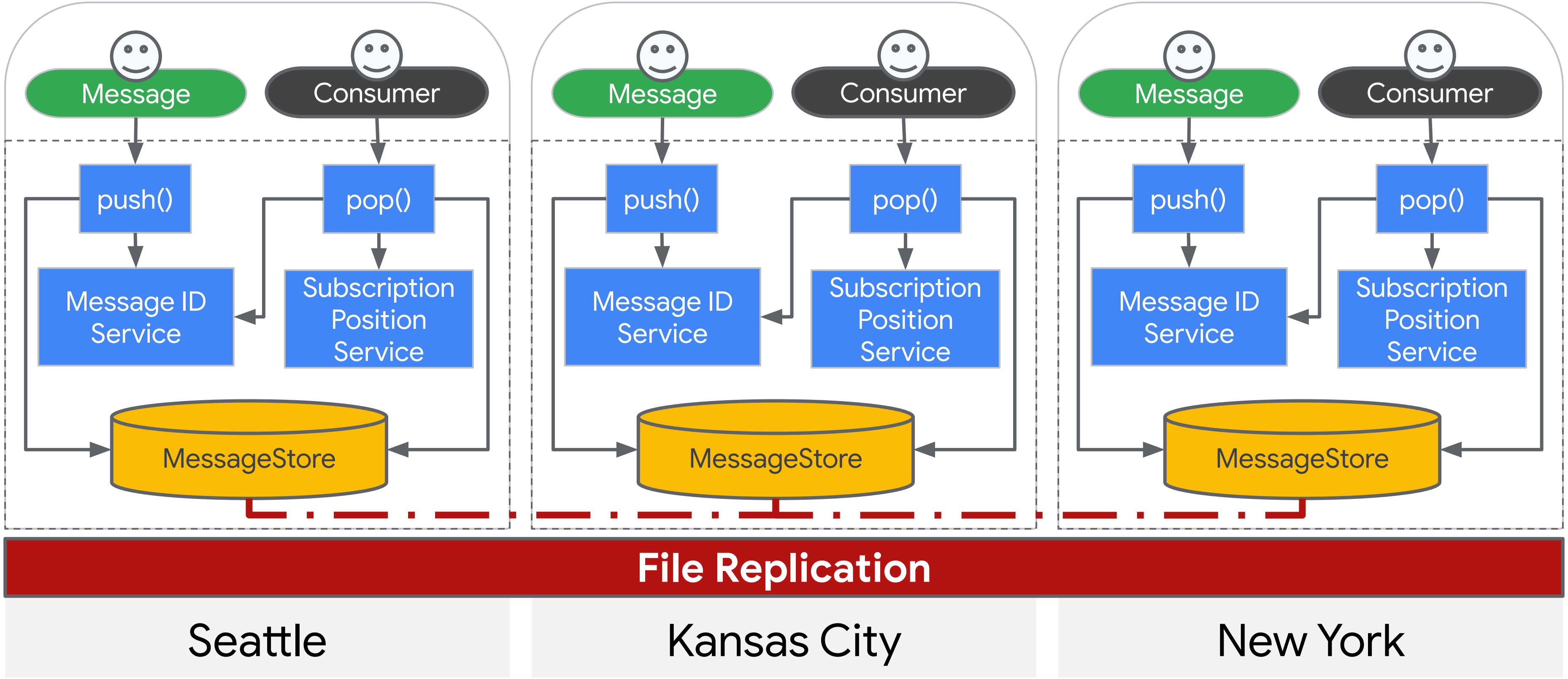
- Each machine failure = lose all in-flight messages = lose ~600 messages
- Machine would have to fail ~8 times / day for us to lose 5k messages (0.01% of incoming messages)

We can afford it!

Reminders:

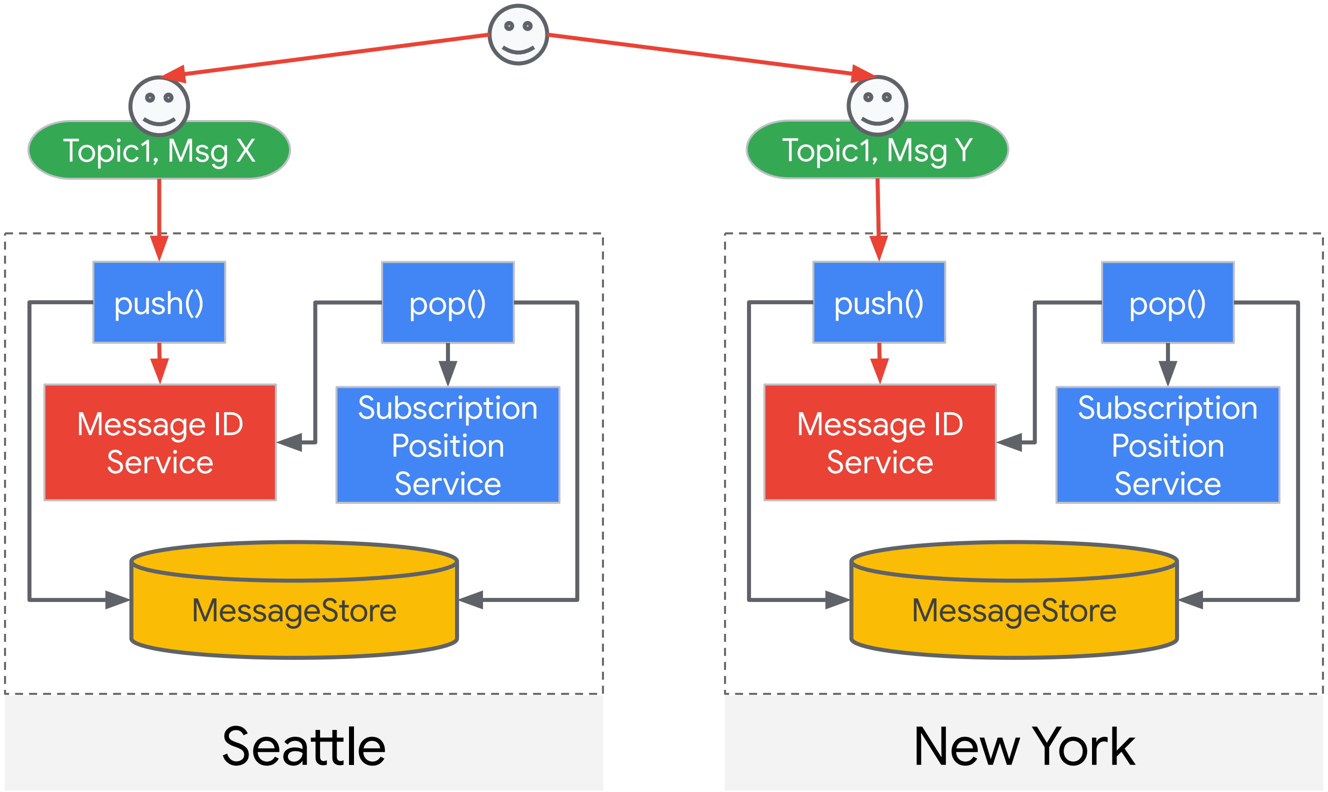
- Can lose 5k msg / day
- ~600 in-flight msg / sec

Let's use replication...

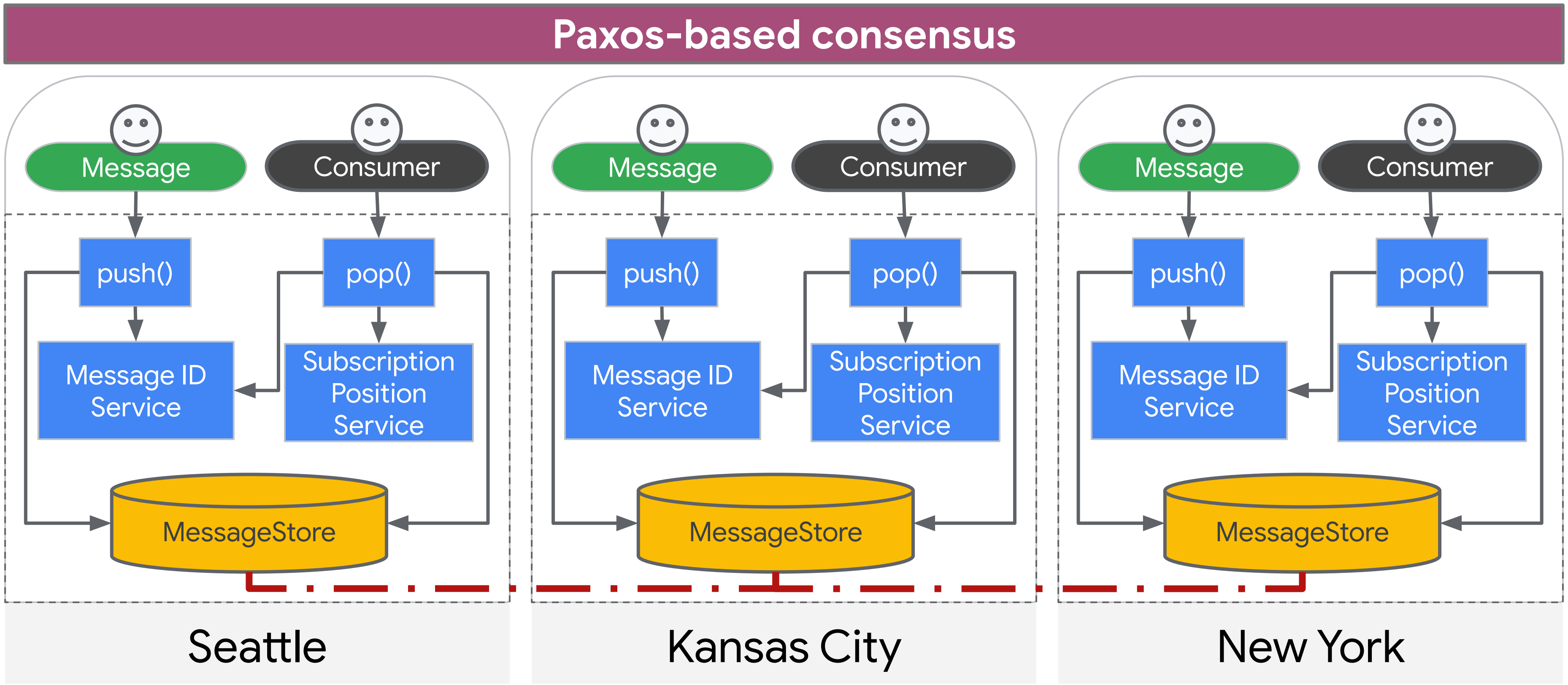


Google

Message ID Conflicts



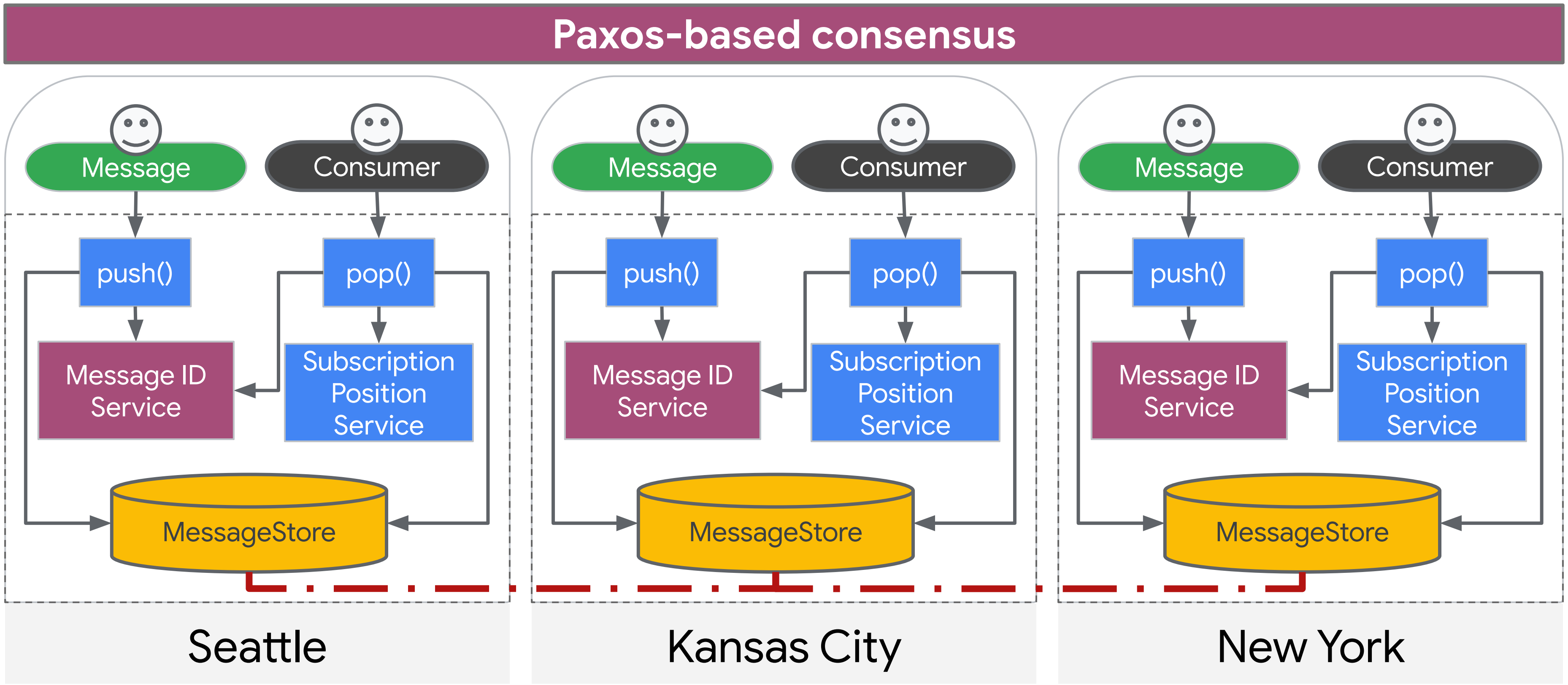
Let's use consensus...



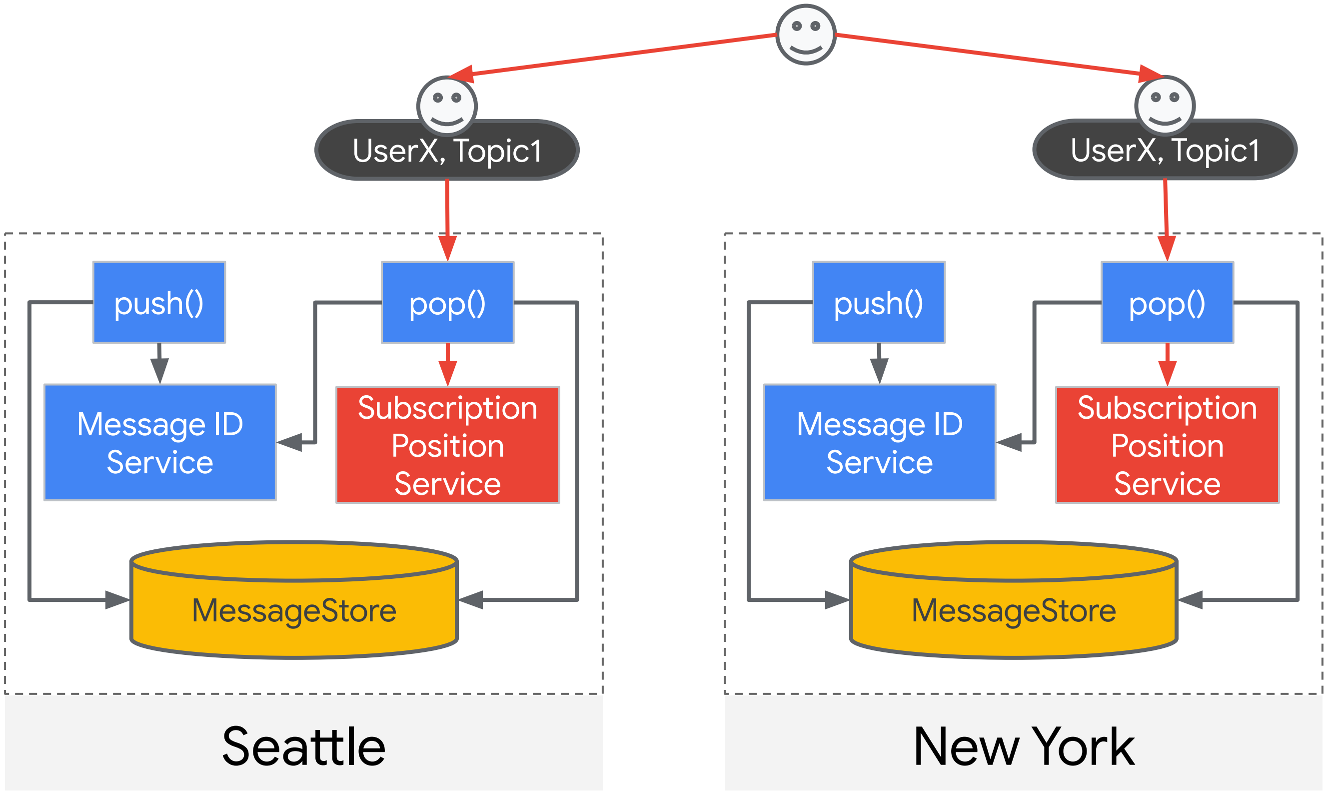
Distributed Consensus

- Distributed components **reliably** and **consistently:**
  - Agree on a single source of truth
  - Identify leaders for specific operations
  - Divide pieces of work
  - Make other decisions
- Unreliable components → **reliable decisions**
- Consistent to decisions, even when sub-components fail
- Recover orphaned datacenters
- Eventual at-most-once semantics
- Paxos, FastPaxos, Raft

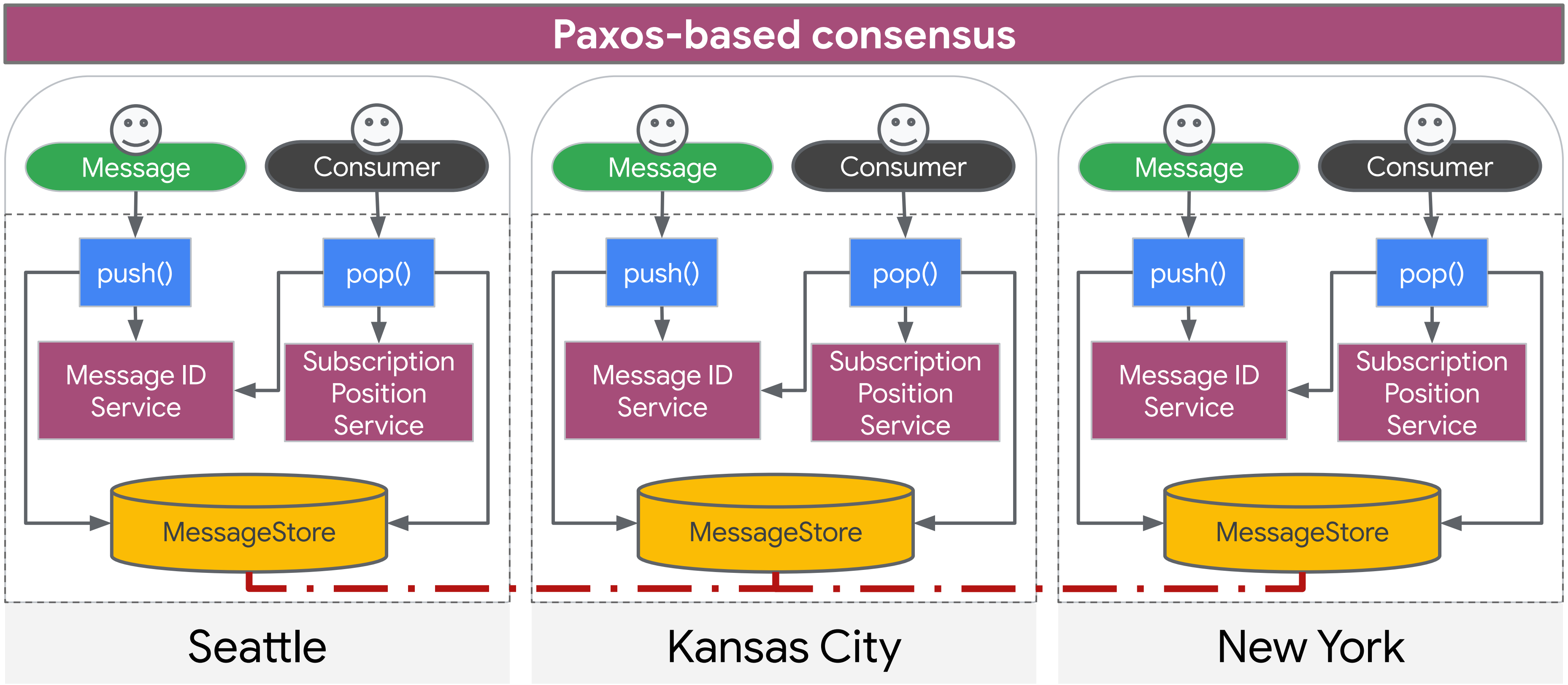
Let's use consensus...



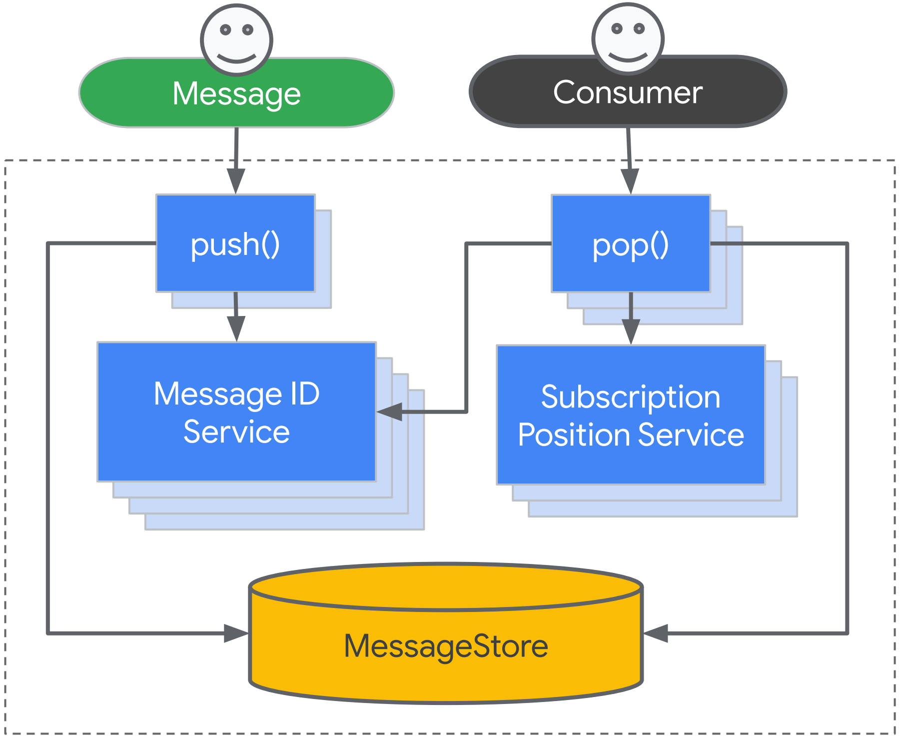
Partitioned/Stale Subscription Positions



Let's use consensus...



Replicating/Sharding Services



Google

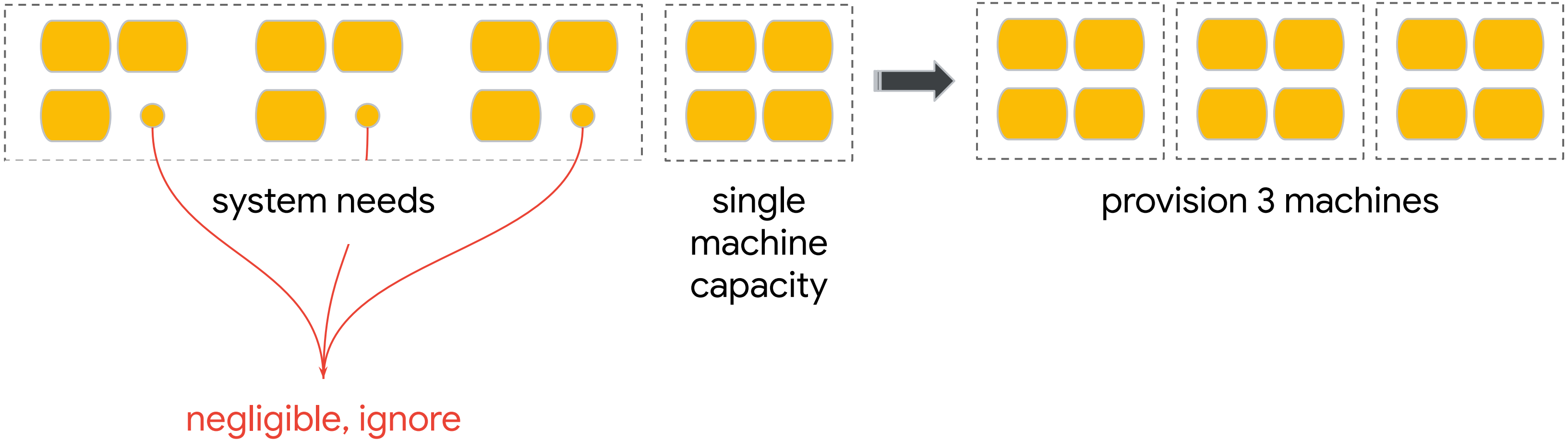
Breakout Session 3:
Provision the System (35 minutes)

Goal:
Identify how many machines you need. Determine if SLOs are viable.

Break: 5 Minutes

Provisioning

- Provisioning is an art.
- Simplify where possible
- Over-provision by default
- Granularity: units of one machine



Google

Storage

Message content:

50M msg / day * 5 kB / msg

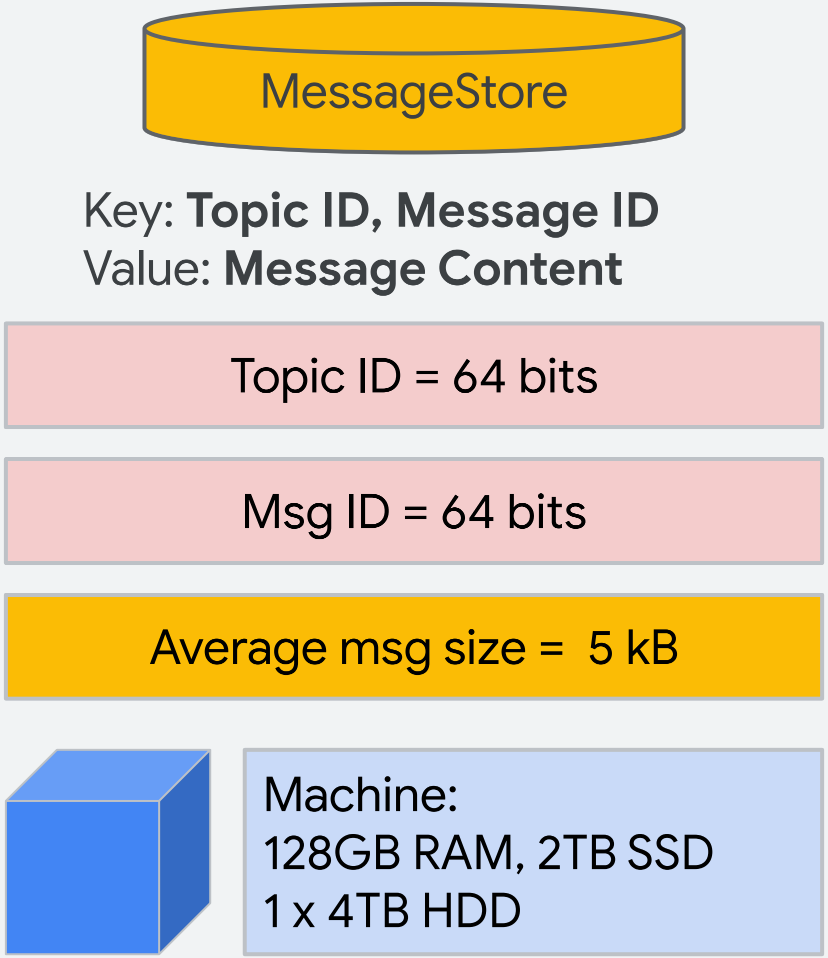
= 250 GB / day

IDs:

50M msg / day * 128 bits / msg

= 800 MB / day

Total: ~250 GB / day



Storage

100 days retention:

250 GB / day * 100 days

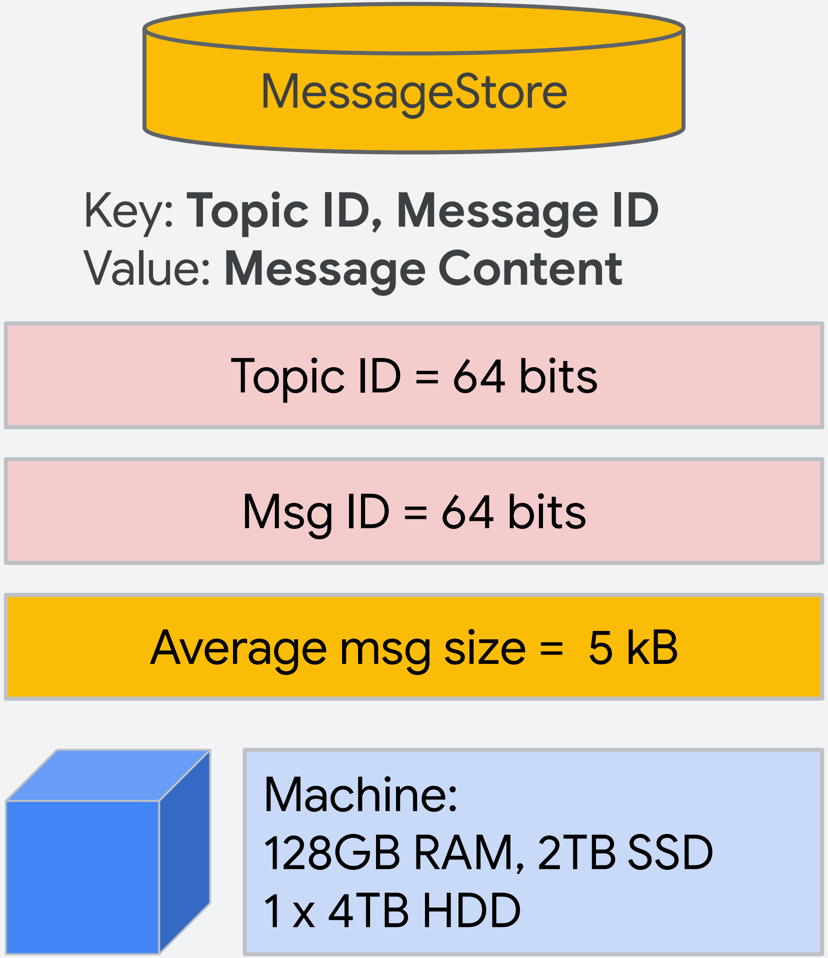
= 25 TB / 100 days

$\lceil 25\text{ TB} / (4\text{ TB HDD / machine}) \rceil$

= 7 machines

... per DC

... per copy



Storage

100 days retention:

7 machines / DC / copy

7 machines / DC / copy

* 2 copies / DC

* 3 DCs

= 42 machines

Key: **Topic ID, Message ID**
Value: **Message Content**

Topic ID = 64 bits

Msg ID = 64 bits

Average msg size = 5 kB

Machine:
128GB RAM, 2TB SSD
1 x 4TB HDD

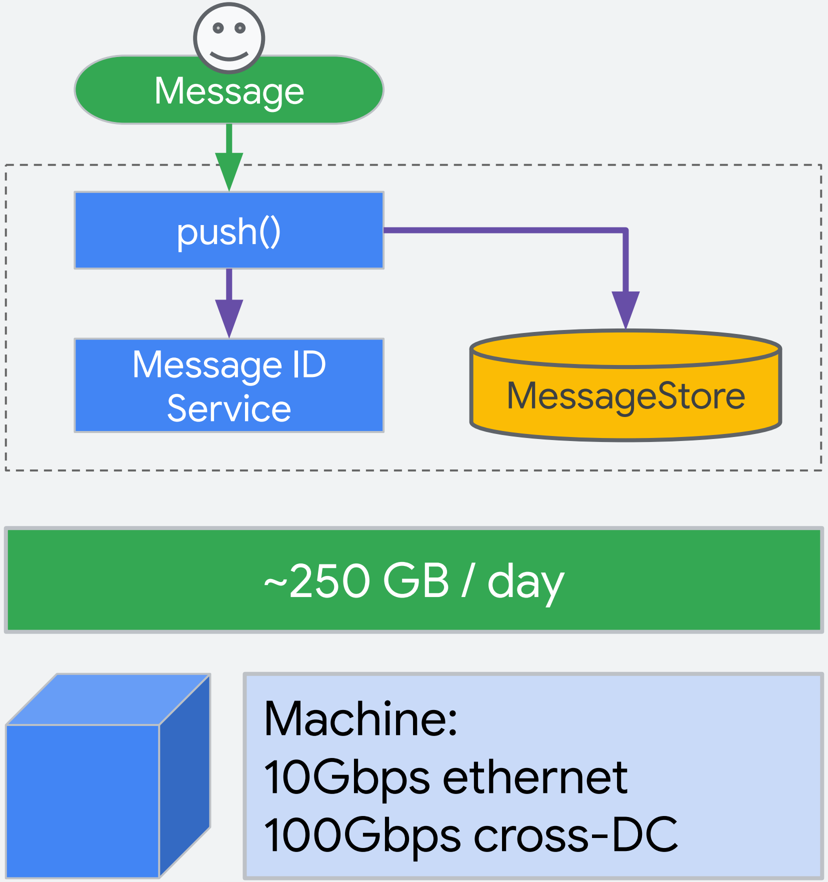
Which hardware to choose?

	latency	per-machine	machine count
RAM	**0.01ms**	**128GB**	**1176**
SSD	**1ms**	**2TB**	**78**
HDD	**15ms**	**4TB**	**42**

Bandwidth: push

- Peak load = 1.25x avg load
  = 250 GB / day * 1.25
  = ~315 GB / day
- 315 GB / day
  = ~4 MB / s
  = ~30 Mbps inbound
- Outbound ~= Inbound

30 Mbps inbound,
30 Mbps outbound



Bandwidth: pop

- Avg load

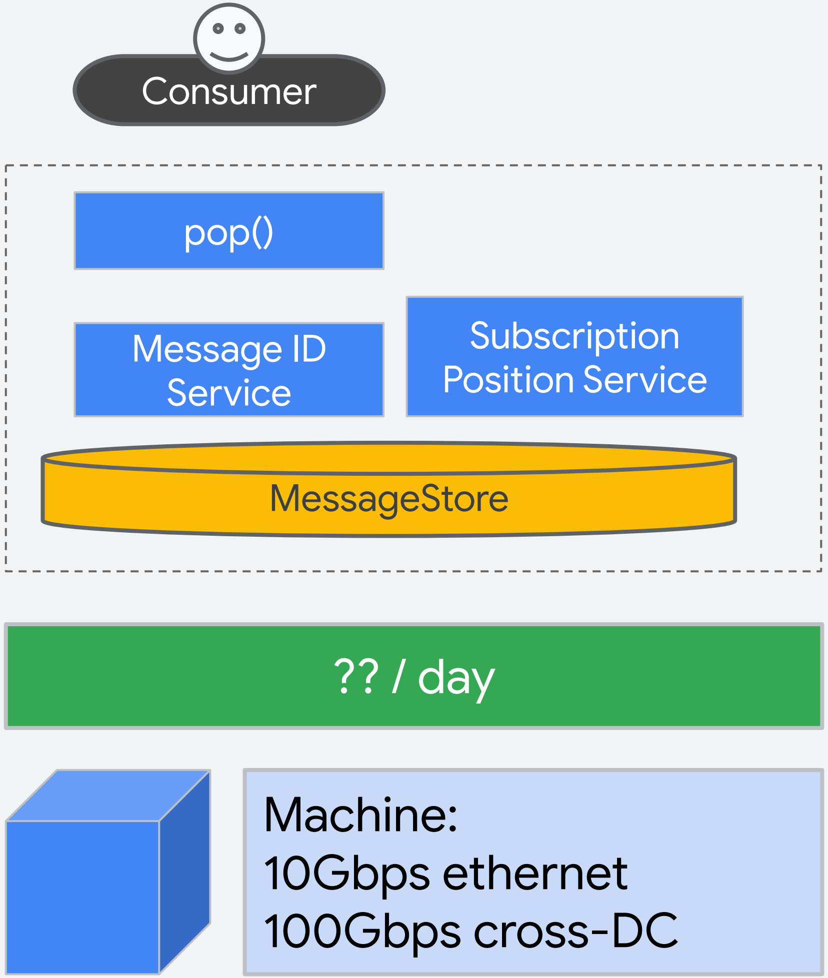
  = 10k consumers *

  5 topics / consumer *

  10k msg / topic / day *

  5 kB / msg

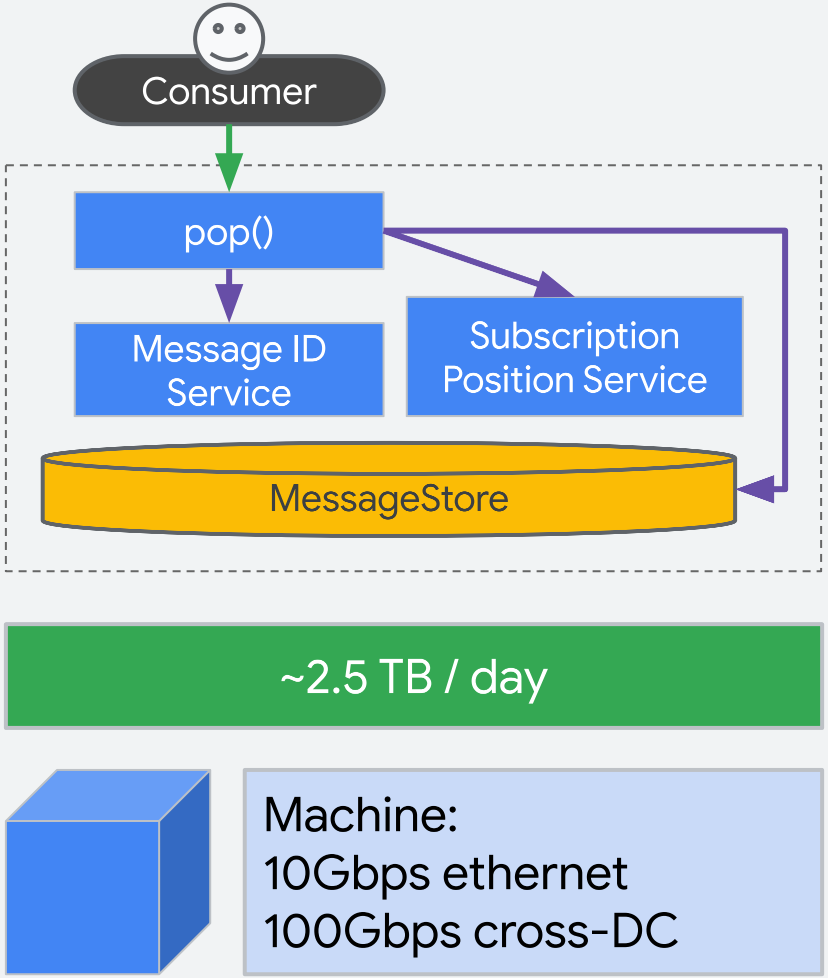
  = 2.5 TB / day



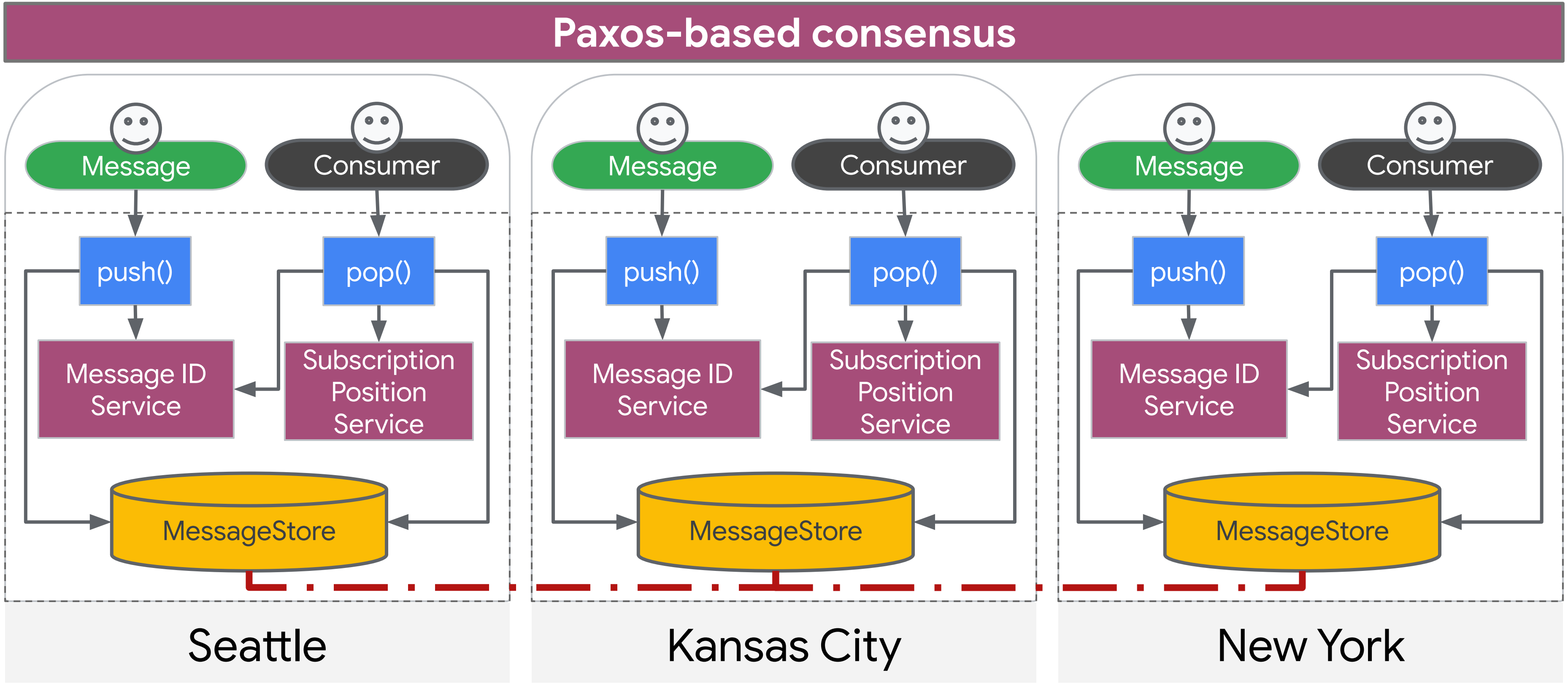
Bandwidth: pop

- Peak load = 1.25x avg load
  = 2.5 TB / day * 1.25
  = ~3.15 TB / day
- 3.15 TB / day
  = ~37 MB / s
  = ~300 Mbps outbound
- Internal ~= Outbound

300 Mbps outbound,
300 Mbps internal

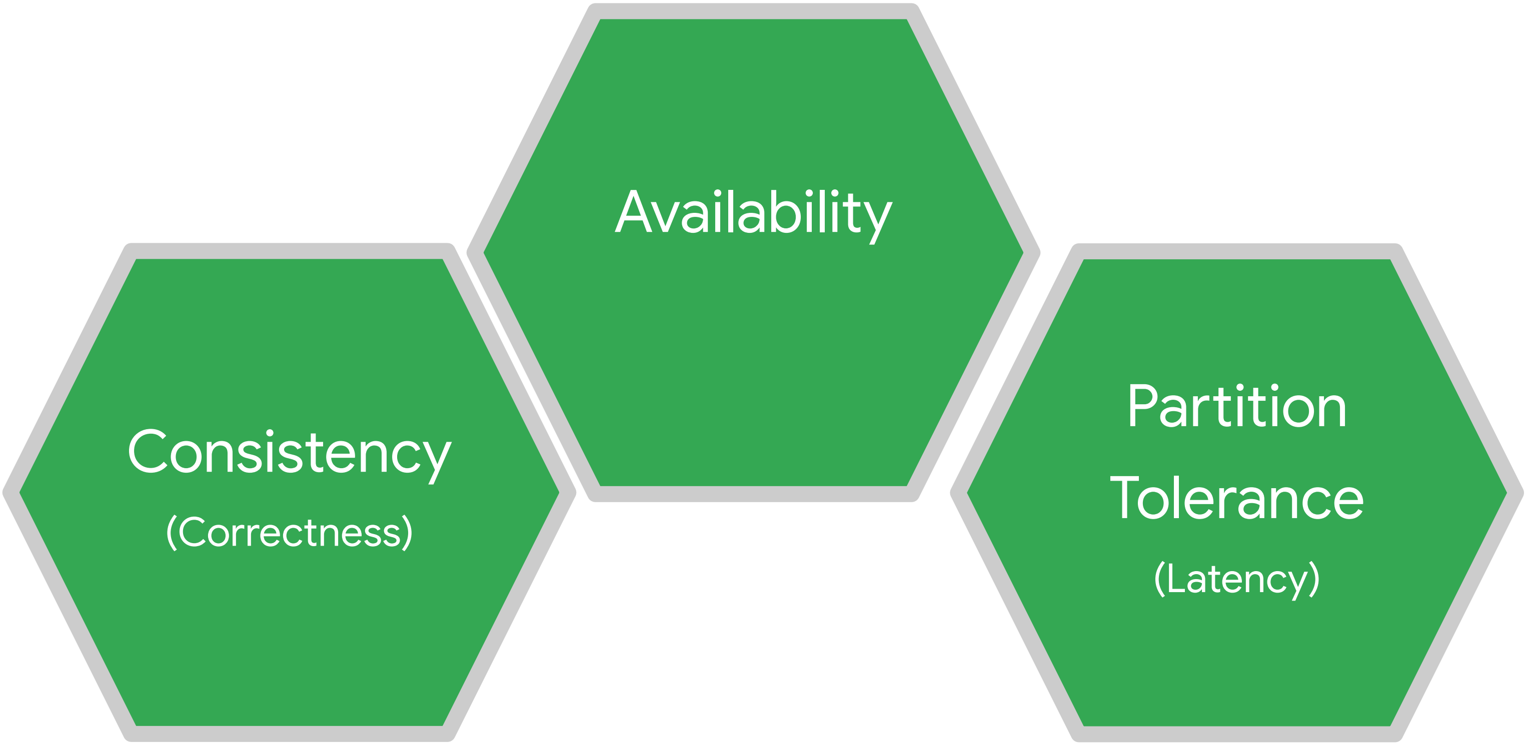


Is it reliable enough?



Google

CAP Theorem


Availability
Consistency
(Correctness)
Partition
Tolerance
(Latency)

Latency: push

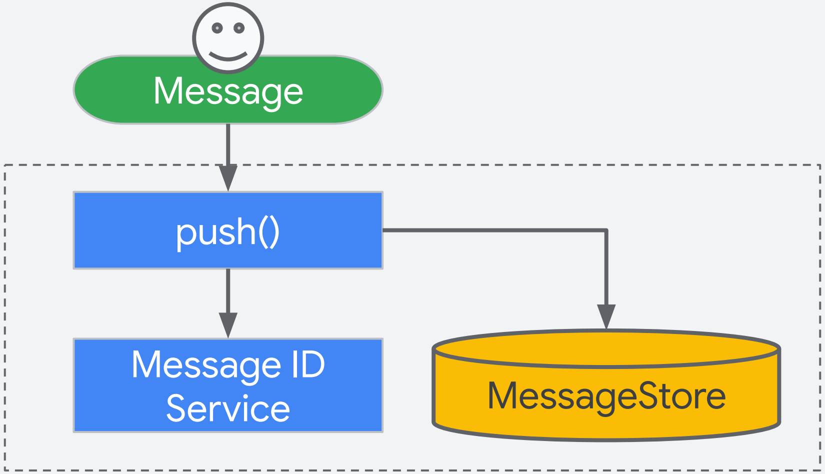
- Determine ID: ~200ms
- Store message: ~150ms
  - Synchronous
  - Bound by slowest connection to remote datacenter
- Write message: ~10ms

Total = 200ms + 150ms + 10ms

= 360ms

Reminders:

- 99% ops complete in <500ms
- Paxos takes ~200ms
- Inter-continental = ~150ms
- Local write takes ~10ms



Latency: pop

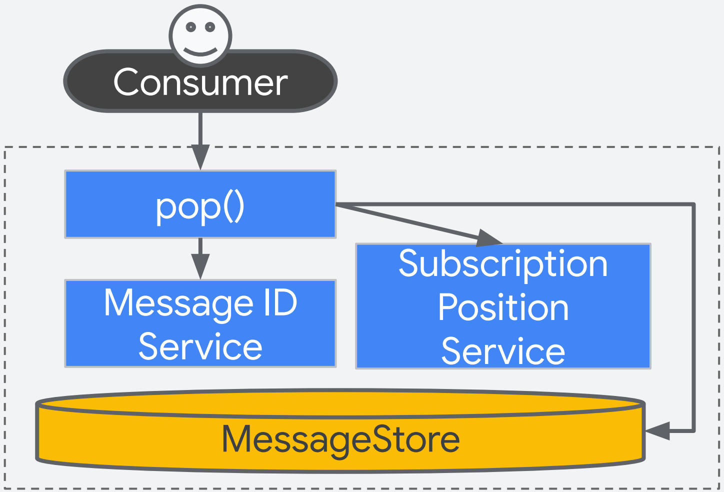
- Determine ID: ~0.5ms local, ~150ms remote
- Read message: ~15ms local, ~150ms remote
- Deliver message: ~negligible
- Update position: ~200ms

Total = 150ms + 150ms + 200ms

= **500ms**

Reminders:

- 99% ops complete in <500ms
- Paxos takes ~200ms
- Inter-continental = ~150ms
- Disk seek+read takes ~15ms



Bill of Materials

Final count of machines:

2 push +

2 pop +

3 Message ID Service +

3 Subscription Position Service +

14 MessageStore

= 24 per DC * 3 DCs * 1.25 (for load spikes)

= 90 machines

Last thoughts

- Start simple and iterate
- See the big picture
- Details, details, details!
- But also, be reasonably pragmatic
- Flexible vs. premature future-proofing
- Cultivate discipline in problem solving approach
- Make data-driven decisions

Take breaks and enjoy the process!

Site Reliability Engineering

Distributed PubSub

Non-Abstract Large System Design