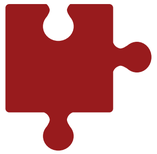
CS 591 K1:
Data Stream Processing and Analytics

Spring 2021

02/02: Stream ingestion and pub/sub systems

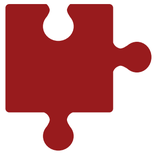
Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

Streaming sources



Where do stream processors read data from?

Streaming sources



Where do stream processors read data from?

Files, e.g. transaction logs

Sockets

IoT devices and sensors

Databases and KV stores

Message queues and brokers

Challenges

Streaming sources…

- can be distributed
  - out-of-sync sources may produce out-of-order streams
- can be connected to the network
  - latency and unpredictable delays
- might be producing too fast
  - stream processor needs to keep up and not shed load
- might be producing too slow or become idle
  - stream processor should be able to make progress
- might fail (or seem as if they failed)

Producers and consumers

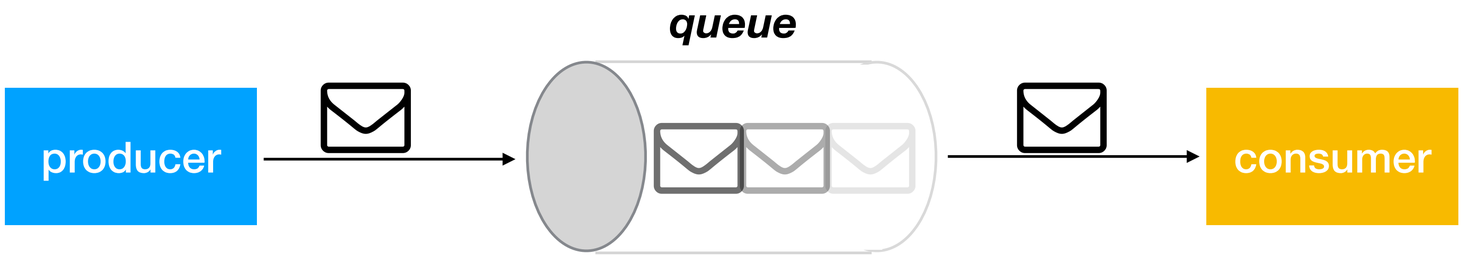
- An event is typically generated by a **producer** (or **publisher** or **sender**) and processed by one or multiple **consumers** (or **subscribers** or **recipients**)
- Events are commonly grouped into the same **topic**
  - in a similar way batch data belonging to the same file are grouped together
  - topics are commonly events of the same **type**: `userCreated`, `userLoggedIn`, `userLoggedOut`, `userSentPayment`

Connecting producers to consumers

- Indirectly
  - Producer writes to a file or database
  - Consumer periodically polls and retrieves new data
    - polling overhead, latency?
  - Consumer receives a notification when new data is available
    - how to implement triggers?
- Direct messaging
  - Direct network communication, UDP multicast, TCP
  - HTTP or RPC if the consumer exposes a service on the network
  - Failure handling: application needs to be aware of message loss, producers and consumers always online

Message queues

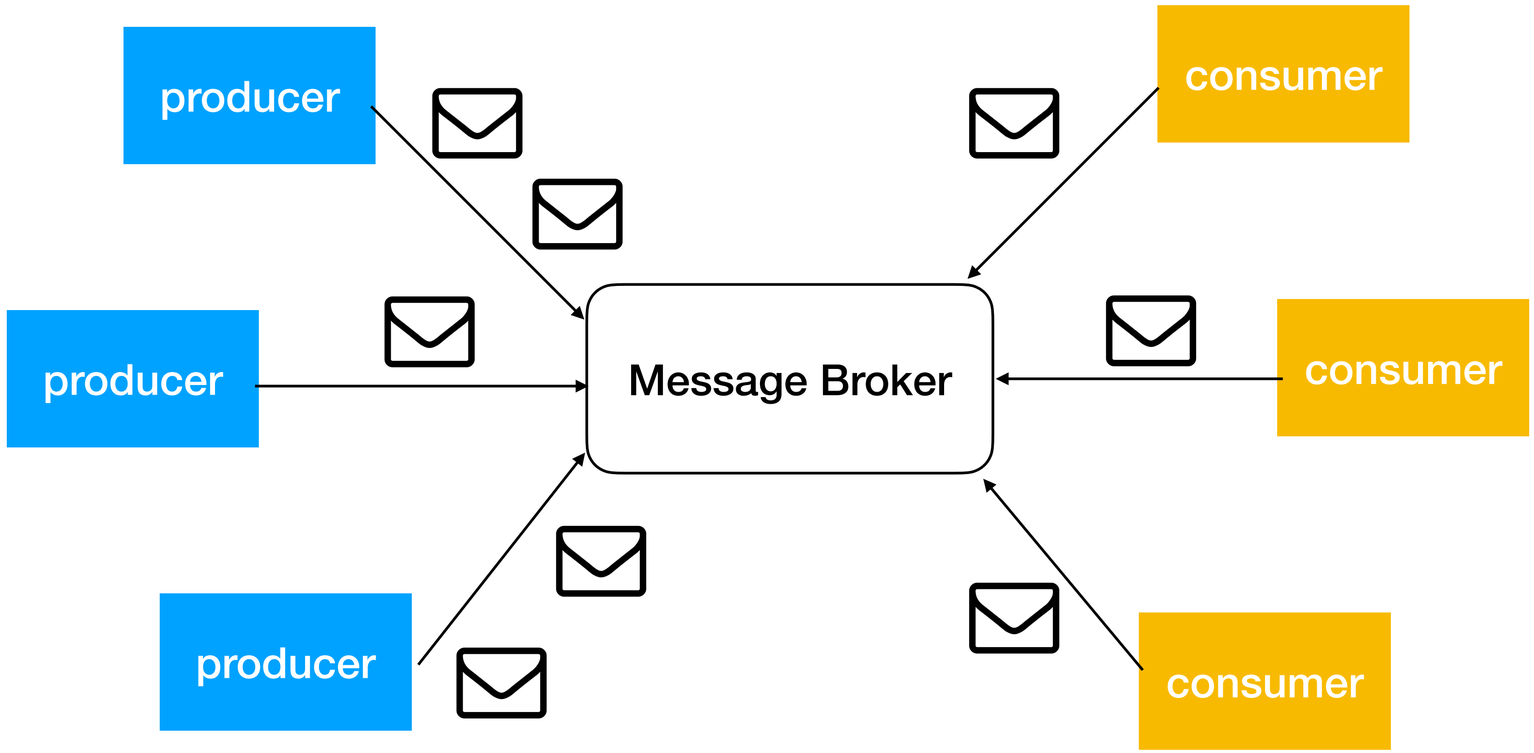
- Asynchronous **point-to-point** communication
- Lightweight buffer for **temporary** storage
- Messages stored on the queue until they are processed and **deleted**
  - transactional, timing, and ordering guarantees
- Each message is processed only *once*, by a single consumer
- Event retrieval is not defined by content / structure but its **order**
  - FIFO, priority



Message brokers

Message broker: a system that connects event producers with event consumers.

- It receives messages from the producers and pushes them to the consumers.
- A TCP connection is a simple messaging system which connects one sender with one recipient.
- A general messaging system connects multiple producers to multiple consumers by organizing messages into topics.



- messages not removed after consumption
- multiple consumers can retrieve the same message
- many-to-many communication
- message content / structure matters for delivery

MB architecture advantages

- Multiple producers/consumers as concurrent clients
- Effective failure handling, crashes or disconnects
- Broker responsible for message durability
- Asynchronous communication, i.e. producer only needs to receive ack from broker

Communication patterns (I)

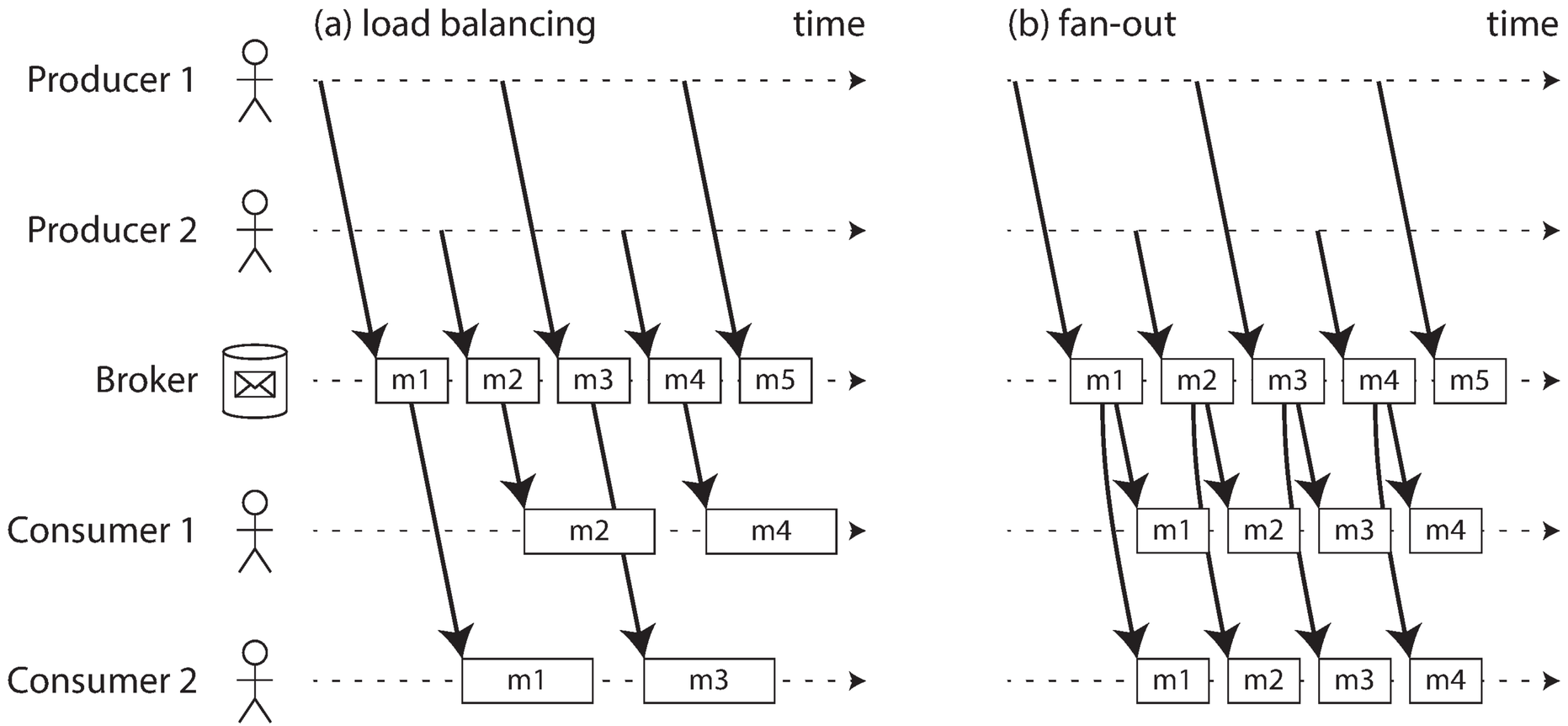
Load balancing or **shared subscription**

- A logical producer/consumer can be implemented by multiple physical tasks running in parallel
- If a producer generates events with high rate, we can balance the load by spawning several consumer processes
- The broker can choose to send messages to consumers in a **round-robin** fashion

Communication patterns (II)

Fan-out

Several logical consumers (possibly implemented by several parallel physical processes) can subscribe to the same topic, so that the message broker delivers messages to all subscribed consumers in a **broadcast** fashion.


(a) load balancing
time
Producer 1
Producer 2
Broker
Consumer 1
Consumer 2
m1
m2
m3
m4
m5
m2
m4
m1
m3
(b) fan-out
time
m1
m2
m3
m4
m5
m1
m2
m3
m4
m1
m2
m3
m4

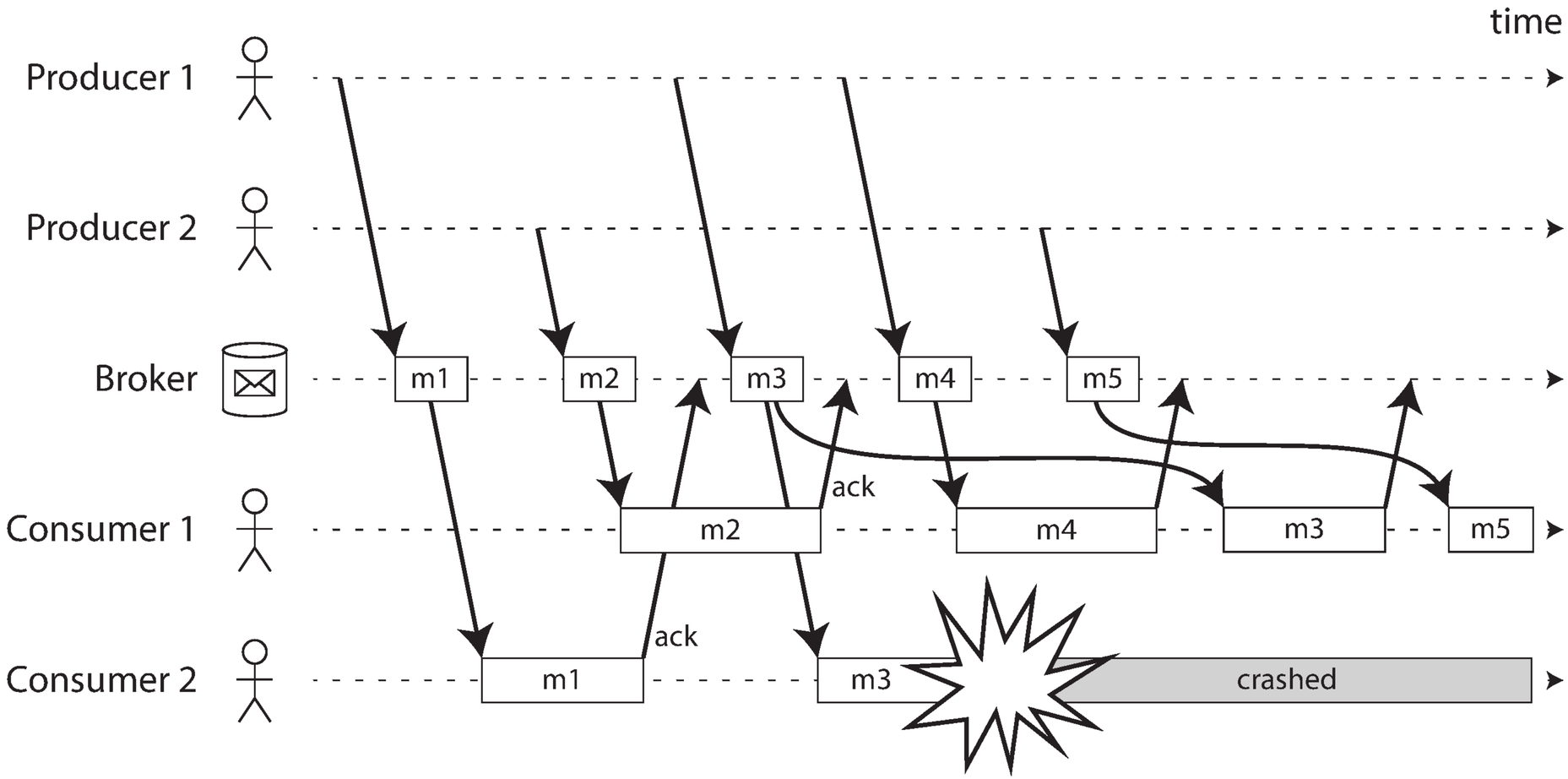
Brokers vs. Databases

- DBs keep data until explicitly deleted while MBs delete messages once consumed.
  - Use a database for long-term data storage!
- MBs assume a small working set. If consumers are slow, throughput might degrade.
- DBs support secondary indexes for efficient search while MBs only offer topic-based subscription.
- DB query results depend on a snapshot and clients are not notified if their query result changes later.

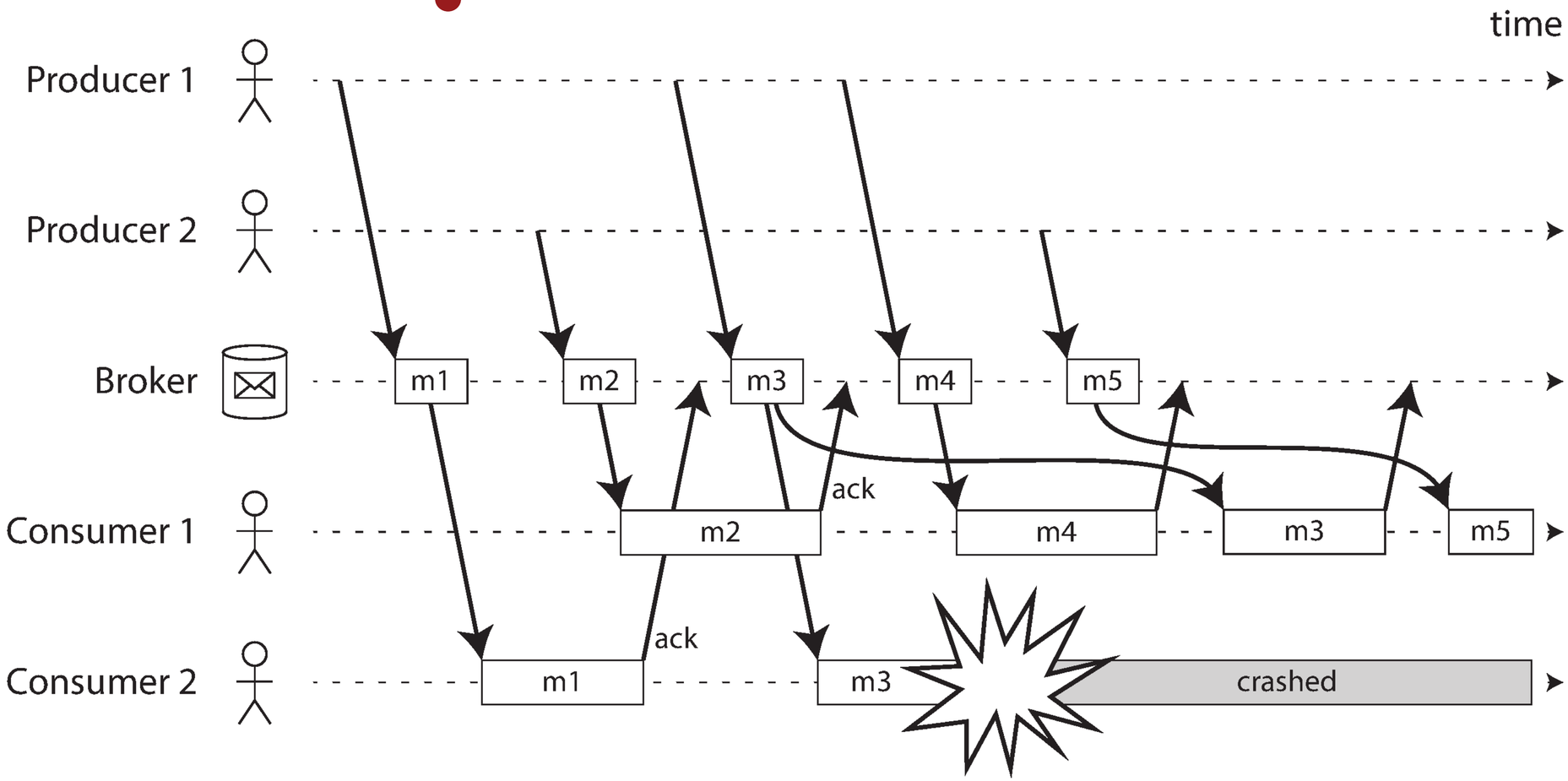
Message delivery and ordering

Acknowledgements are messages from the client to the broker indicating that the client has finished processing a message

- If an acknowledgement is not received, delivery is retried
- Re-delivery might cause re-ordering of messages
- Re-delivery complicates stream processing and fault-tolerance
  - might process a message out-of-order or twice

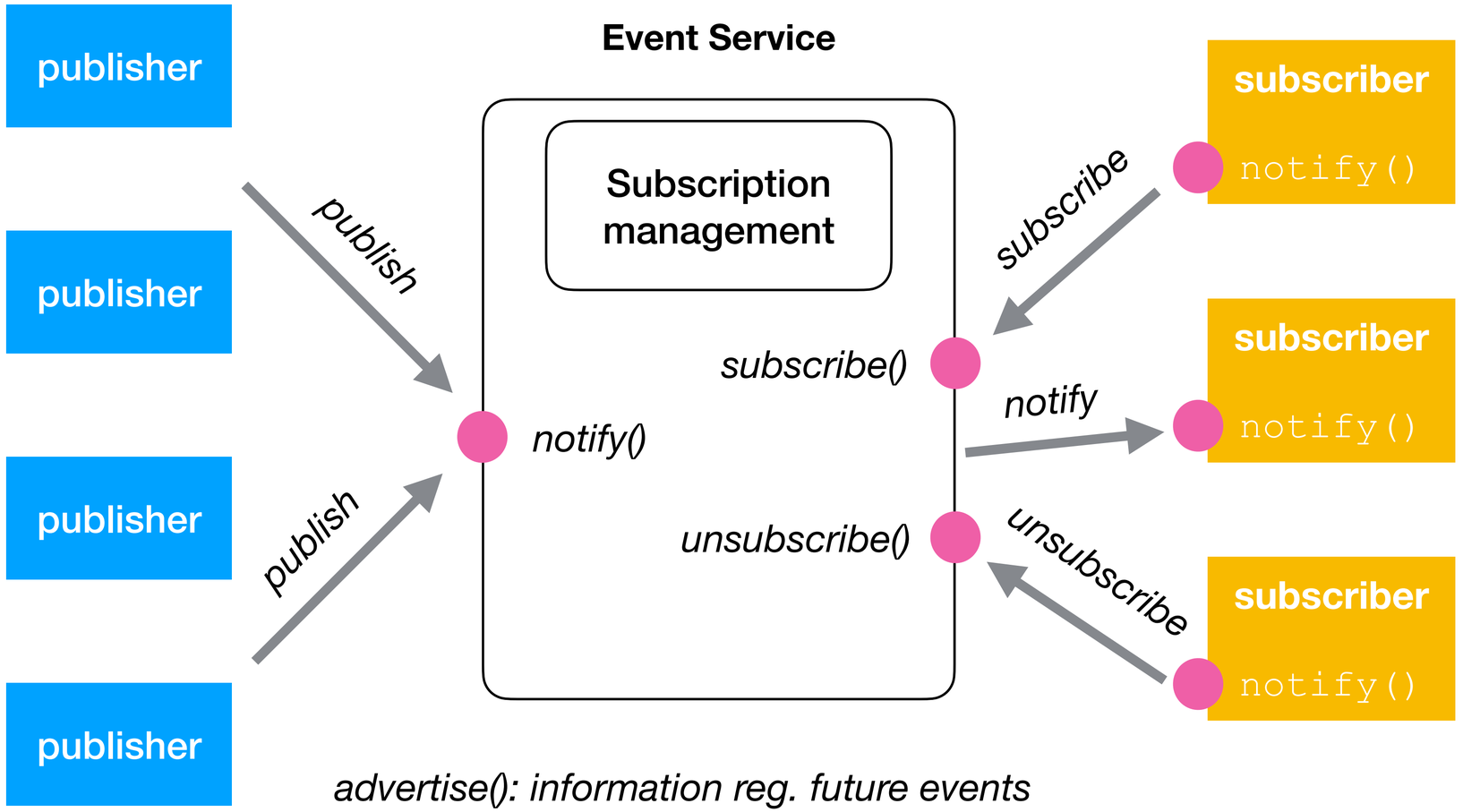

time
Producer 1
Producer 2
Broker
Consumer 1
Consumer 2
m1
m2
m3
m4
m5
ack
m2
m4
m3
m5
ack
m1
m3
crashed

How can we avoid this?



Publish/Subscribe Systems

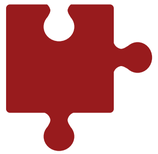
Publish/Subscribe Systems



advertise(): information reg. future events

Pub/Sub levels of de-coupling

- **Space**: interacting parties do not need to know each other
  - Publishers do not know who / how many subscribers there are.
- **Time**: interacting parties do not need to actively participate in the interaction at the same time
  - Publishers can produce events when subscribers are disconnected.
- **Synchronization**: interacting parties are not blocked
  - Subscribers get notified asynchronously while possibly performing some other concurrent action.



Can you fill this in?

Paradigm	Space Decoupling	Time Decoupling	Synchronization Decoupling
Message-passing			
RPC/RMI			
Asynchronous RPC			
Futures			
Message Queues			
Pub/Sub	Yes	Yes	Yes

Pub/Sub vs. other paradigms

Paradigm	Space Decoupling	Time Decoupling	Synchronization Decoupling
Message-passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC	No	No	Yes
Futures	No	No	Yes
Message Queues	Yes	Yes	Producer-side
Pub/Sub	Yes	Yes	Yes

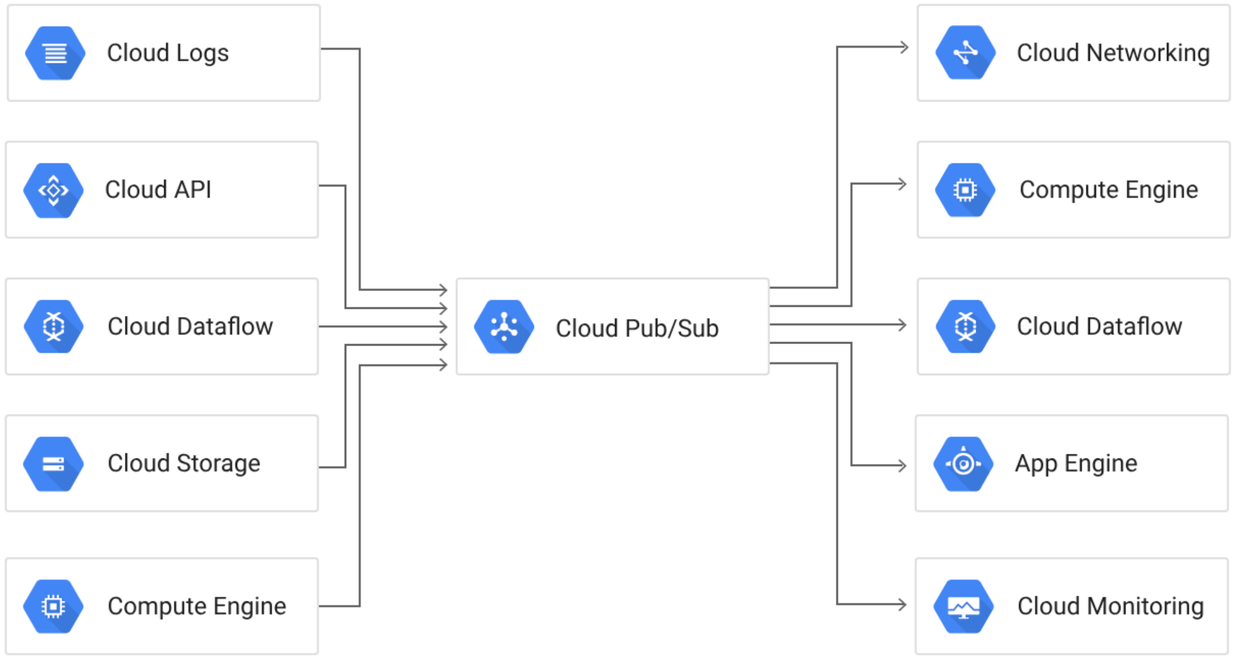
Topic-based Pub/Sub

- Events are grouped into **topics** which are identified by *keywords*.
- Topics <—> Groups
  - Subscribing to a topic T can be viewed as becoming a member of a **group** T.
  - Publishing an event on topic T can be viewed as *broadcasting* the event to all members of group T.
- Topic *hierarchies* allow topic organization according to containment relationships.
  - subscribing to a topic implicitly involves subscribing to all sub-topics of that topic, too.
- Topic names are represented with URL-like notation and some systems also allow the use of wildcards.

Content-based Pub/Sub

- Events are grouped according to event **properties** or **contents**.
  - data attributes or meta-data.
- Consumers subscribe to events by specifying *filters* in a subscription language.
- Filters define **constraints** in the form of name-value pairs and basic comparison operators.
- Constraints can be logically combined to form complex event patterns.
  - company == ‘Uber’ and price < 100
- Predecessors of Complex Event Processing (CEP) systems

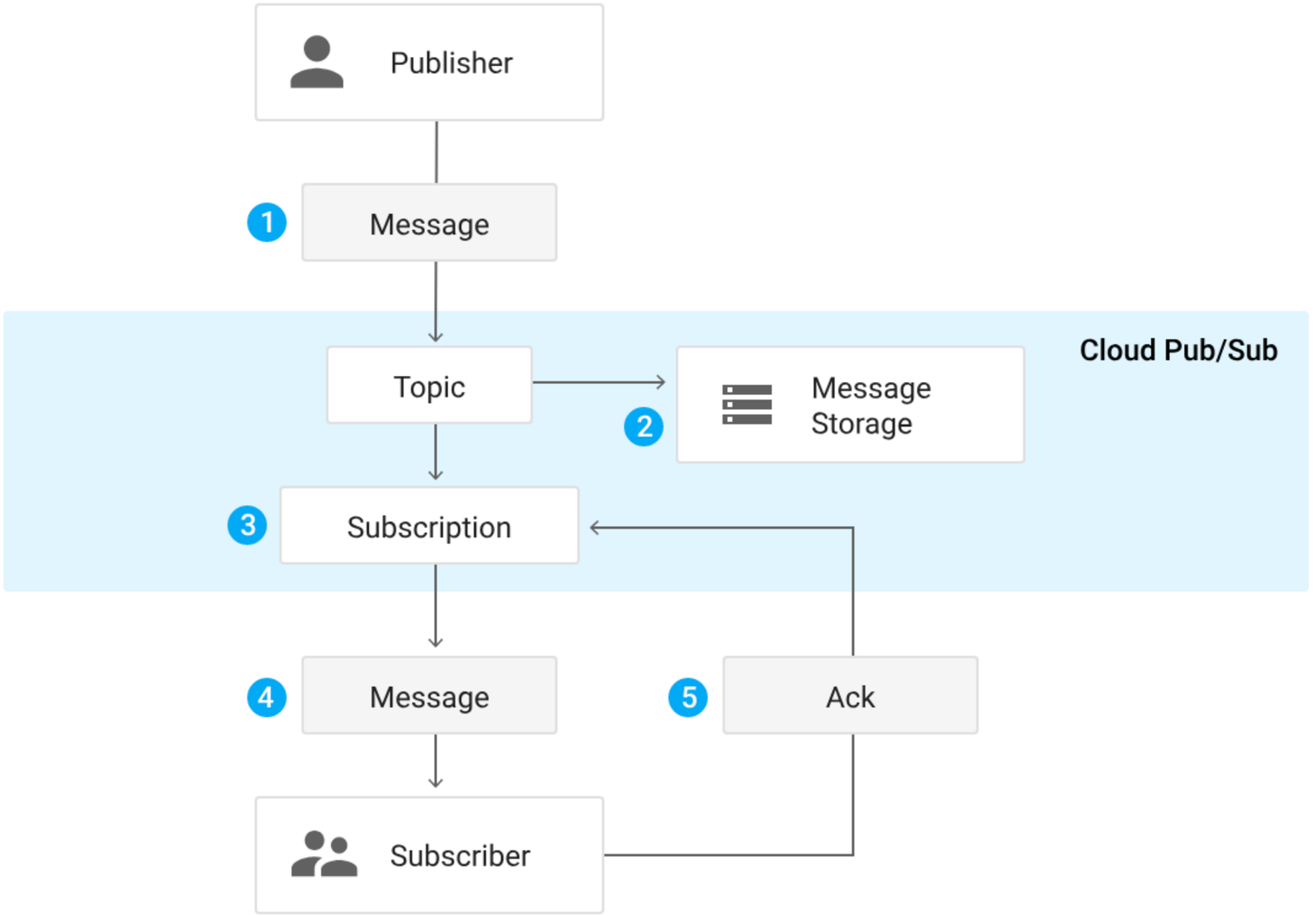
Google Cloud Pub/Sub

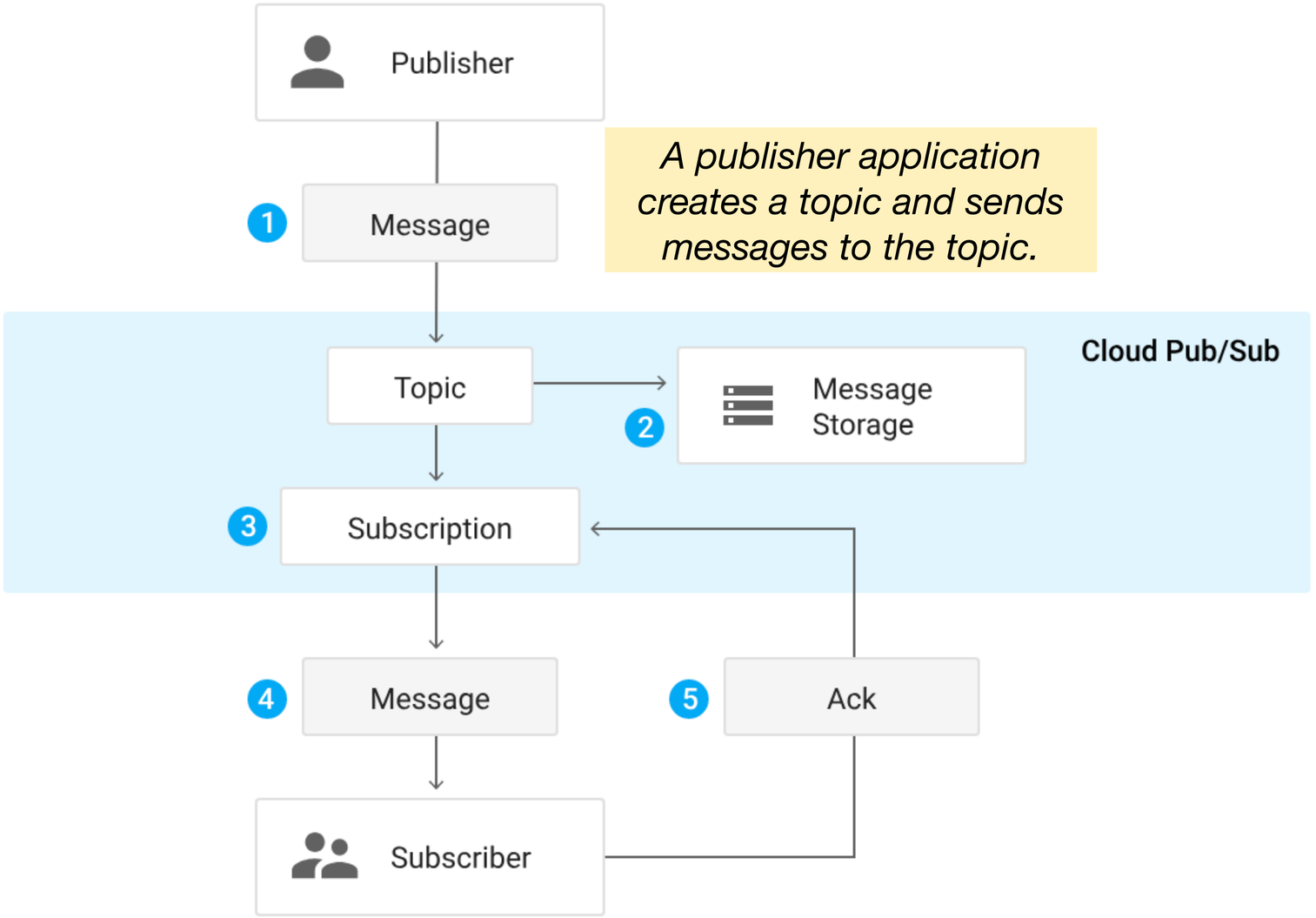


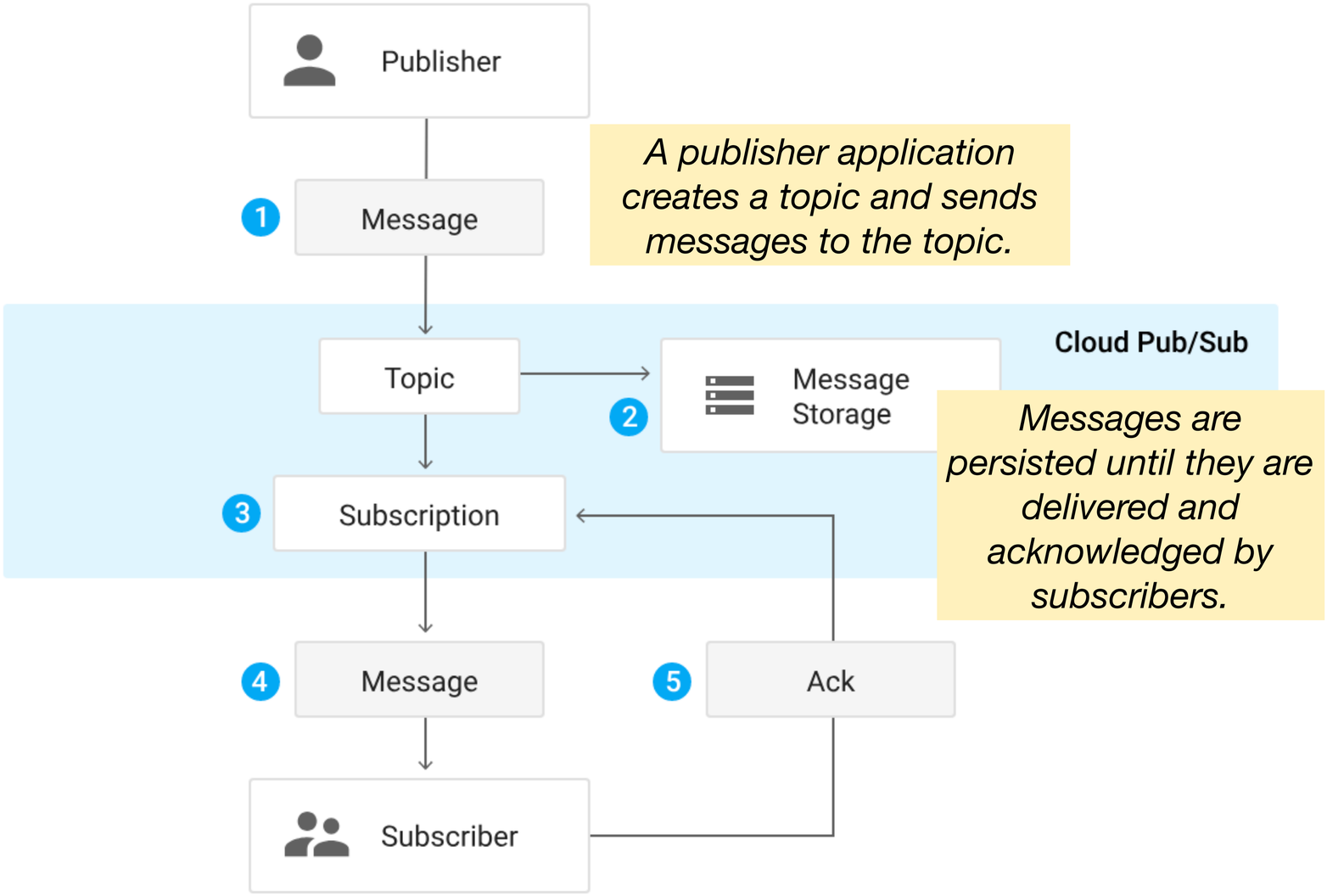
Publishers and Subscribers are applications.

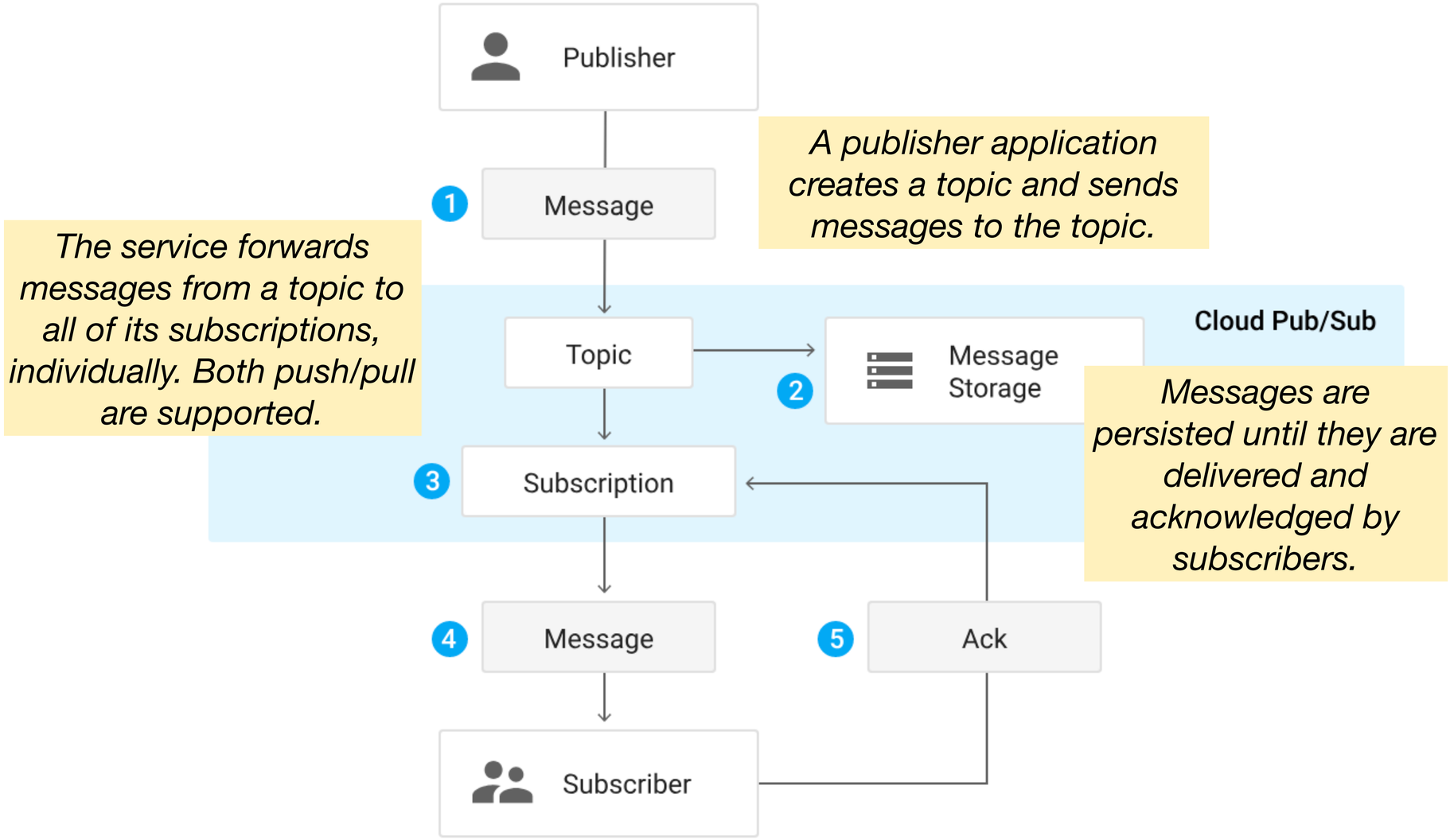
Use-cases

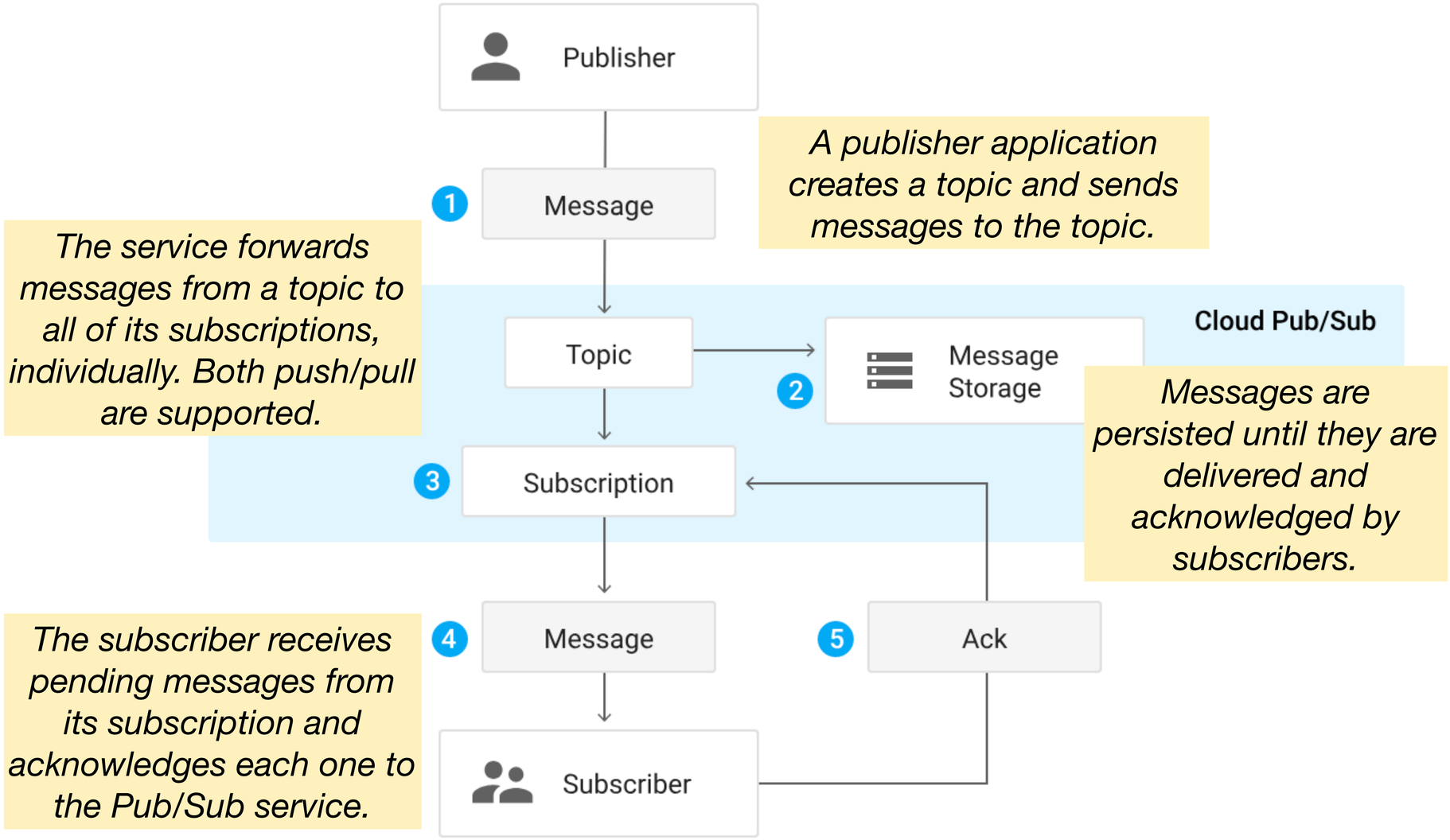
- **Balancing workloads in network clusters**
  - tasks can be efficiently distributed among multiple workers, such as Google Compute Engine instances.
- **Distributing event notifications**
  - a service that accepts user signups can send notifications whenever a new user registers, and downstream services can subscribe to receive notifications of the event.
- **Refreshing distributed caches**
  - an application can publish invalidation events to update the IDs of objects that have changed.
- **Logging to multiple systems**
  - a Google Compute Engine instance can write logs to the monitoring system, to a database for later querying, and so on.
- **Data streaming from various processes or devices**
  - a residential sensor can stream data to backend servers hosted in the cloud.

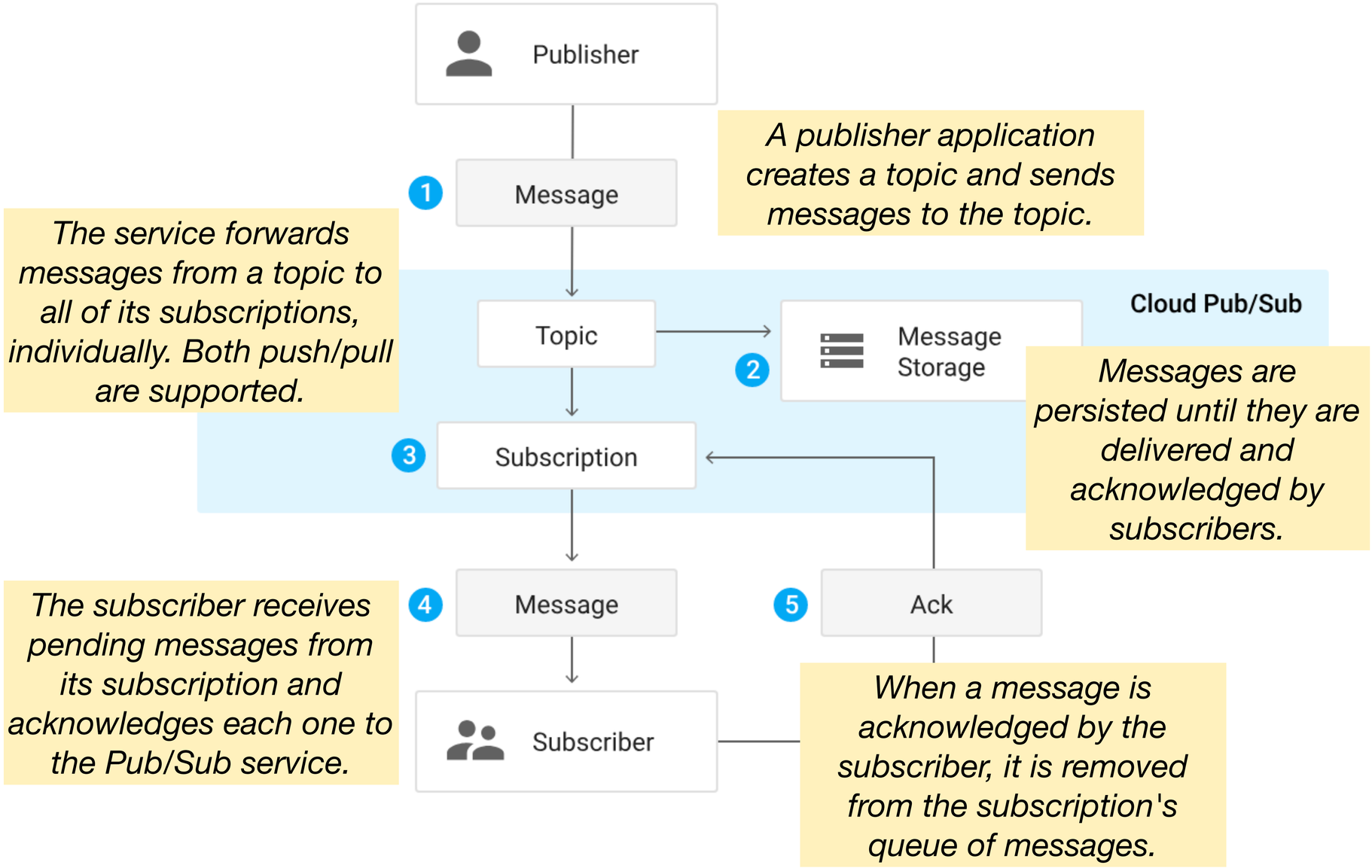

Publisher
1
Message
Cloud Pub/Sub
Topic
2
Message Storage
3
Subscription
4
Message
5
Ack
Subscriber


Publisher
1
Message
A publisher application creates a topic and sends messages to the topic.
Cloud Pub/Sub
Topic
2
Message Storage
3
Subscription
4
Message
5
Ack
Subscriber


Publisher
1
Message
A publisher application creates a topic and sends messages to the topic.
Cloud Pub/Sub
Topic
2
Message Storage
Messages are persisted until they are delivered and acknowledged by subscribers.
3
Subscription
4
Message
5
Ack
Subscriber


Publisher
1
Message
A publisher application creates a topic and sends messages to the topic.
The service forwards messages from a topic to all of its subscriptions, individually. Both push/pull are supported.
Cloud Pub/Sub
Topic
2
Message Storage
Messages are persisted until they are delivered and acknowledged by subscribers.
3
Subscription
4
Message
5
Ack
Subscriber


Publisher
1
Message
A publisher application creates a topic and sends messages to the topic.
The service forwards messages from a topic to all of its subscriptions, individually. Both push/pull are supported.
Cloud Pub/Sub
Topic
2
Message Storage
Messages are persisted until they are delivered and acknowledged by subscribers.
3
Subscription
4
Message
5
Ack
The subscriber receives pending messages from its subscription and acknowledges each one to the Pub/Sub service.
Subscriber


Publisher
1
Message
A publisher application creates a topic and sends messages to the topic.
The service forwards messages from a topic to all of its subscriptions, individually. Both push/pull are supported.
Cloud Pub/Sub
Topic
2
Message Storage
Messages are persisted until they are delivered and acknowledged by subscribers.
3
Subscription
4
Message
5
Ack
The subscriber receives pending messages from its subscription and acknowledges each one to the Pub/Sub service.
Subscriber
When a message is acknowledged by the subscriber, it is removed from the subscription's queue of messages.

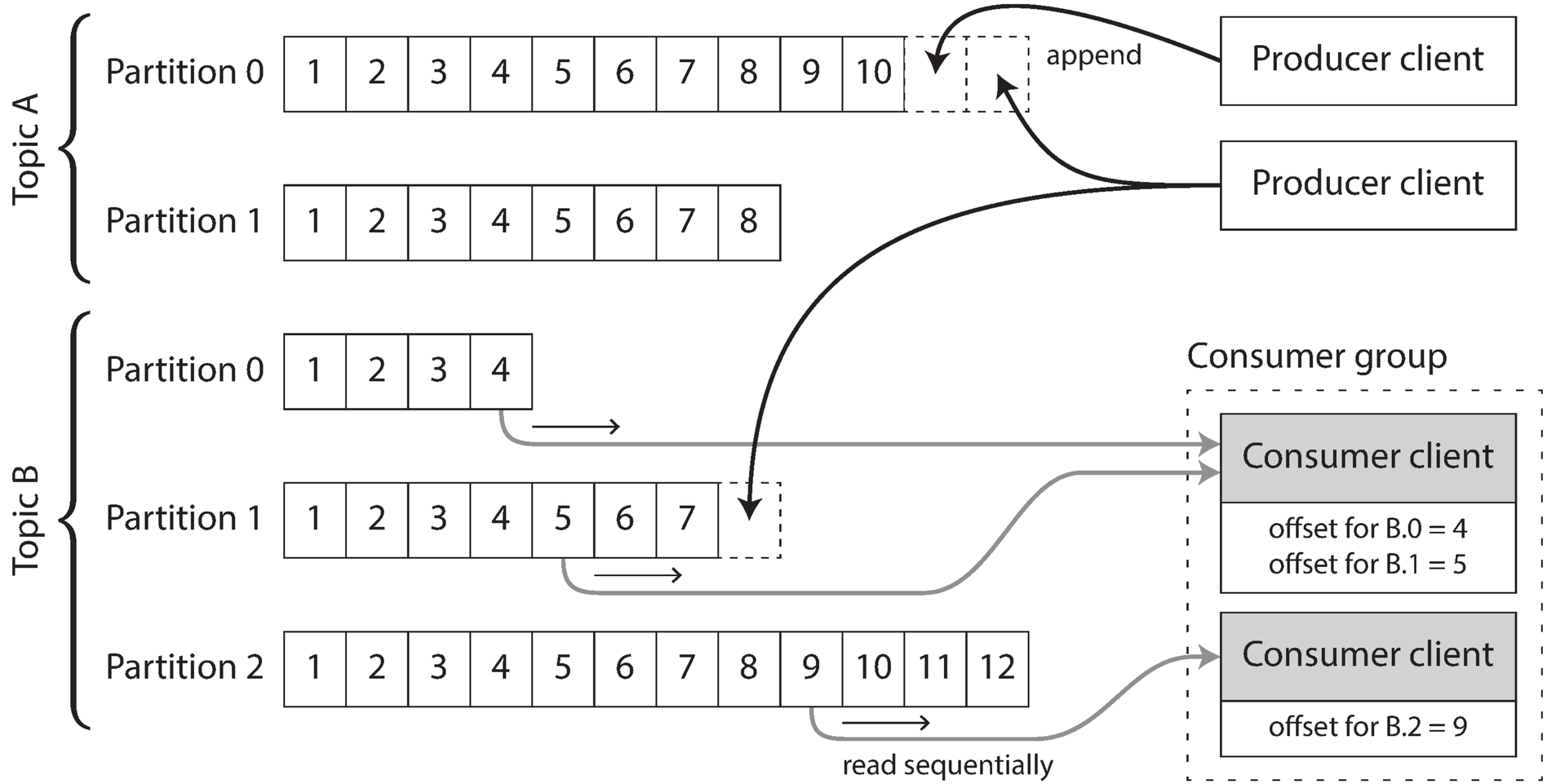
Log-structured brokers

Logs as message brokers

- In typical message brokers, once a message is consumed it is deleted
- Log-based message brokers take a different approach and **durably** store all events in a sequential (possibly partitioned) log
- A **log** is an append-only sequence of records on disk
  - a producer generates messages by simply appending them to the log and a consumer receives messages by reading the log sequentially

Partitions and offsets

- A log can be partitioned, so that each partition can be read and written *independently* of others
  - a topic is a set of partitions
- Within each partition, every message carries an **offset**, a monotonically increasing sequence number
- Within a partition, all messages are *totally ordered* but there is **no ordering guarantee across partitions**


Topic A
Partition 0
1 2 3 4 5 6 7 8 9 10
Partition 1
1 2 3 4 5 6 7 8
append
Producer client
Producer client
Topic B
Partition 0
1 2 3 4
Partition 1
1 2 3 4 5 6 7
Partition 2
1 2 3 4 5 6 7 8 9 10 11 12
Consumer group
Consumer client
offset for B.0 = 4
offset for B.1 = 5
Consumer client
offset for B.2 = 9
read sequentially

Failure handling

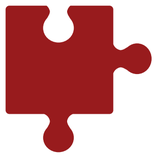
- The broker does not need to wait for acknowledgements any more, but simply **record consumers' offsets periodically**

- If a consumer fails, a new one can take over starting from the last recorded offset of the failed consumer

- This might cause **re-processing** of messages if the failed consumer had read messages later than its recorded offset

Failure handling

- The broker does not need to wait for acknowledgements any more, but simply **record consumers' offsets periodically**
- If a consumer fails, a new one can take over starting from the last recorded offset of the failed consumer
- This might cause **re-processing** of messages if the failed consumer had read messages later than its recorded offset



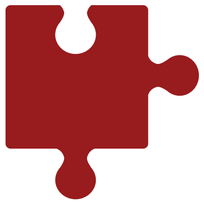
How can we avoid re-processing?

Logs vs. in-memory brokers

- **Multiple consumers with different processing speeds**: reading a message doesn't delete it
- **Coarse-grained load balancing**: assign different partitions to different consumers
- **Limits on maximum parallelism**: the number of the topic's partitions
- **Processing delays**: If a message is slow to process, this delays processing of subsequent messages, as each partition is read by a single thread

Logs vs. in-memory brokers

- **Multiple consumers with different processing speeds**: reading a message doesn't delete it
- **Coarse-grained load balancing**: assign different partitions to different consumers
- **Limits on maximum parallelism**: the number of the topic's partitions
- **Processing delays**: If a message is slow to process, this delays processing of subsequent messages, as each partition is read by a single thread



What would you use when priority is:
- **latency but not ordering?**
- **throughput and ordering?**

How long to keep the log?

- **Log compaction**: a (usually background) process that searches for log records with the same key and merges the records by only keeping the most recent value for each key.

- A key can also be completely removed if it is assigned a special value (tombstone).

A distributed and fault-tolerant publish-subscribe messaging system and serves as the ingestion, storage, and messaging layer for large production streaming pipelines.

Kafka is commonly deployed on a cluster of one or more servers.

Terminology

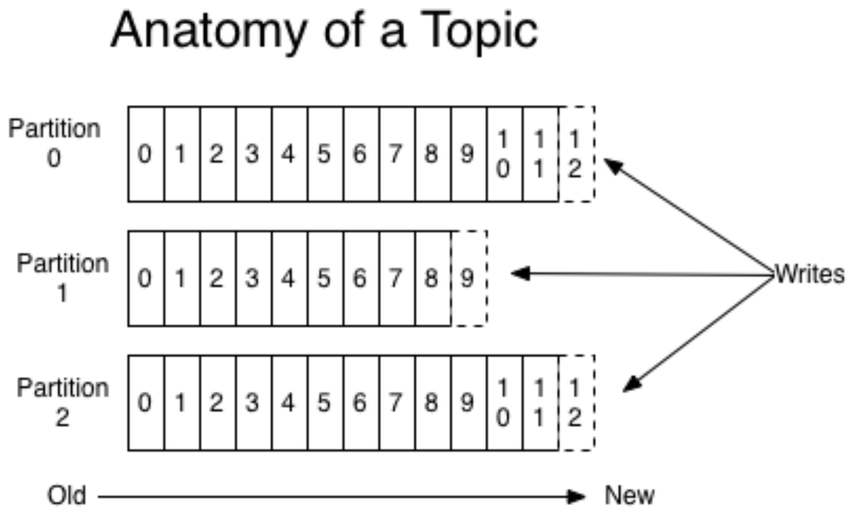
A **topic** identifies a category of stream records stored in a Kafka cluster.
Records consist of a key, a value, and a timestamp.

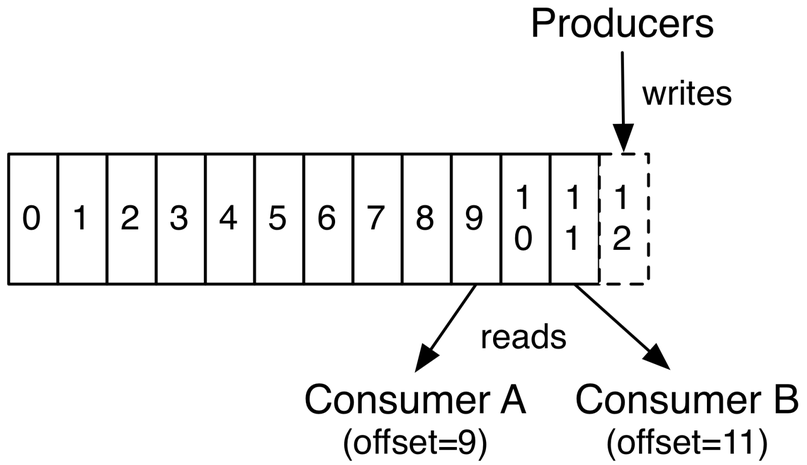
A **producer** publishes a stream of records to a Kafka topic and a **consumer** subscribes to one or more topics and processes the stream of records published in them.

Topics are multi-subscriber, i.e. a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the Kafka cluster maintains a **partitioned log**. Each **partition** is an ordered, immutable sequence of records that is continually appended to—a structured commit log.

An **offset** is a sequential id number assigned to records within a partition. It uniquely identifies records within each partition.

The **retention policy** defines a time period after a record is published that it is available for consumption. Records are discarded after their retention time to free up disk space.

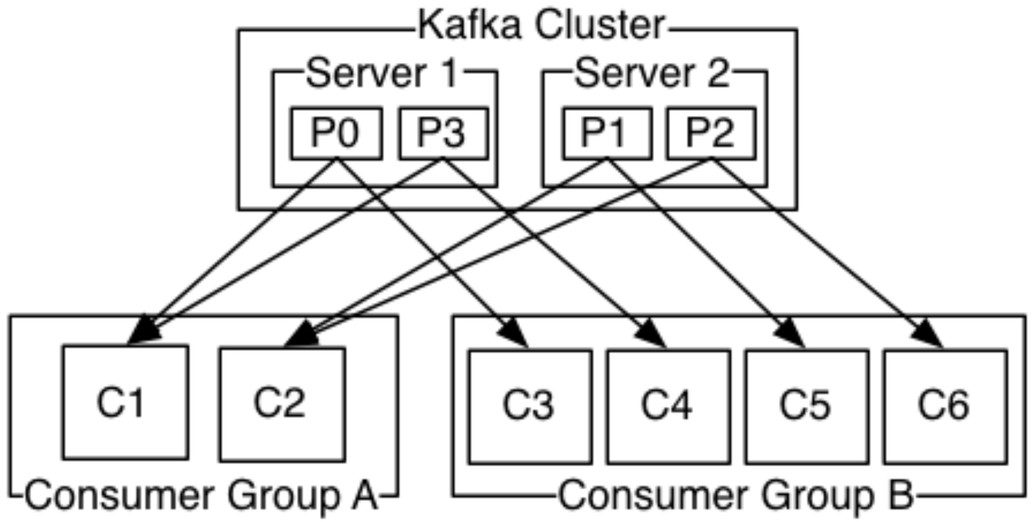




Partitions allow the log to scale beyond a size that will fit on a single server.

Each individual partition must fit on the servers that host it, but a topic may have many partitions so it can handle an arbitrary amount of data.

The number of partitions determines the unit of parallelism.



Consumers label themselves with a *consumer group* name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group.

Consumer instances can be in separate processes or on separate machines.

If all the consumer instances have the same consumer group, then the records will effectively be **load balanced** over the consumer instances.

If all the consumer instances have different consumer groups, then each record will be **broadcast** to all the consumer processes.

Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent:
  - if a record M1 is sent by the same producer as a record M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees records in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any records committed to the log.

Lecture references

The material in this lecture was assembled from the following sources:

- Martin Kleppmann. **Designing data-intensive applications** (O'Reilly Media)

- Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. **The many faces of publish/subscribe**. ACM Comput. Surv. 35, 2 (June 2003)

- https://cloud.google.com/pubsub/docs/overview

Kafka Resources

- Documentation
  - https://kafka.apache.org/
- Community
  - https://kafka.apache.org/contact
- Conference
  - https://kafka-summit.org/