Distributed Systems

Data Streaming

Thoai Nam

High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Engineering

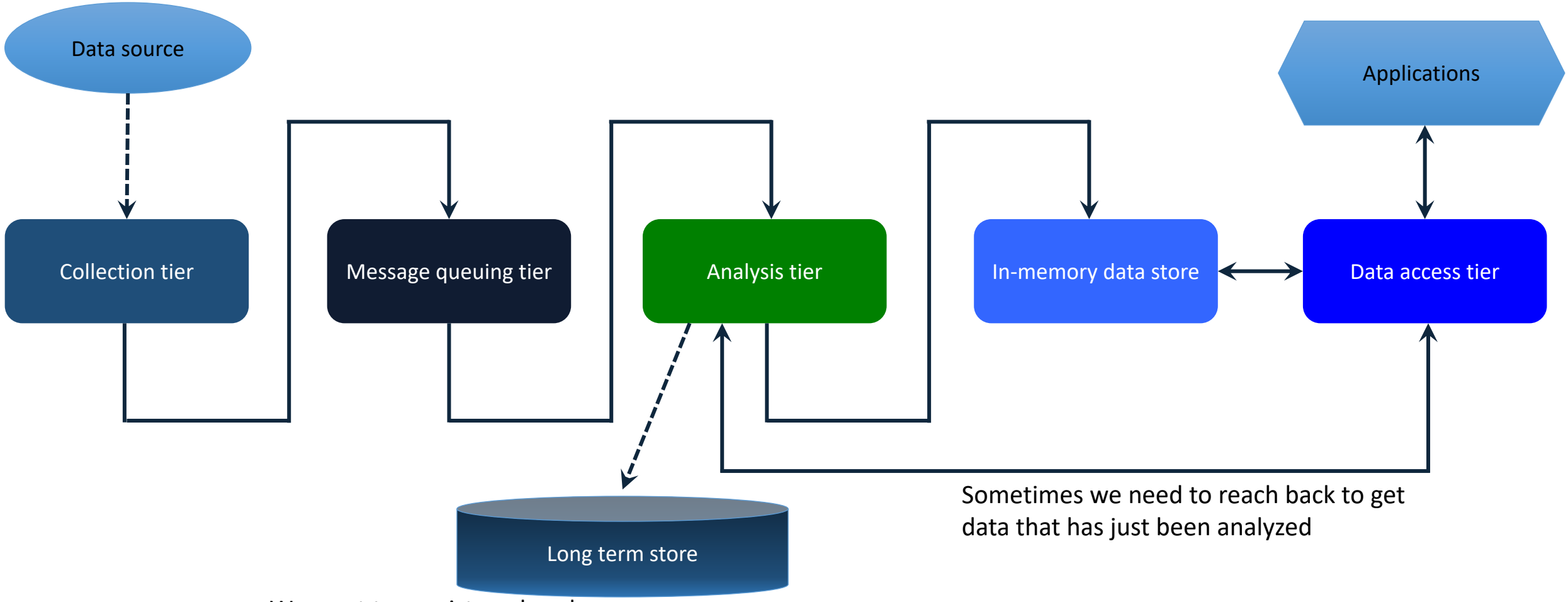
HCMC University of Technology

Classification of real-time systems

Classification	Examples	Latency measured in	Tolerance for delay
Hard	Pacemaker, anti-lock brakes	Microseconds–milliseconds	None—total system failure, potential loss of life
Soft	Airline reservation system, online stock quotes, VoIP (Skype)	Milliseconds–seconds	Low—no system failure, no life at risk
Near	Skype video, home automation	Seconds–minutes	High—no system failure, no life at risk

Data streaming???

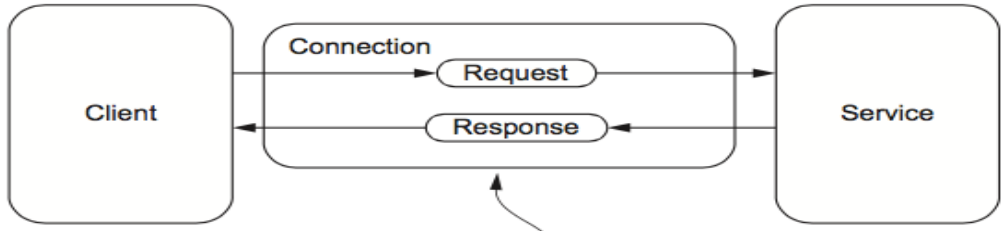
Data streaming architecture



Data collection: data ingestion

- Request/response
- Publish/subscribe
- One-way
- Request/acknowledge
- Stream

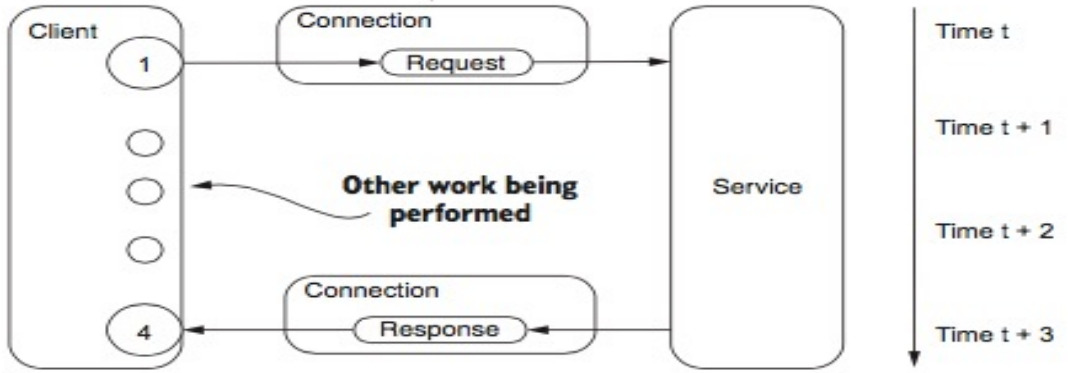
Request/response



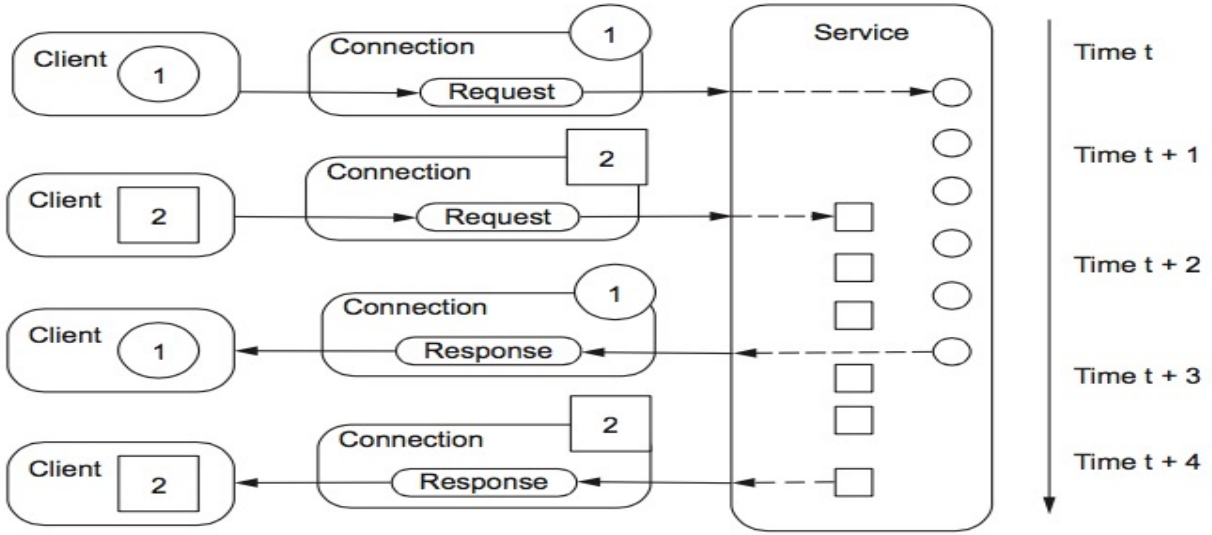
(a) Sync RR

The request and the response happen over the same connection.

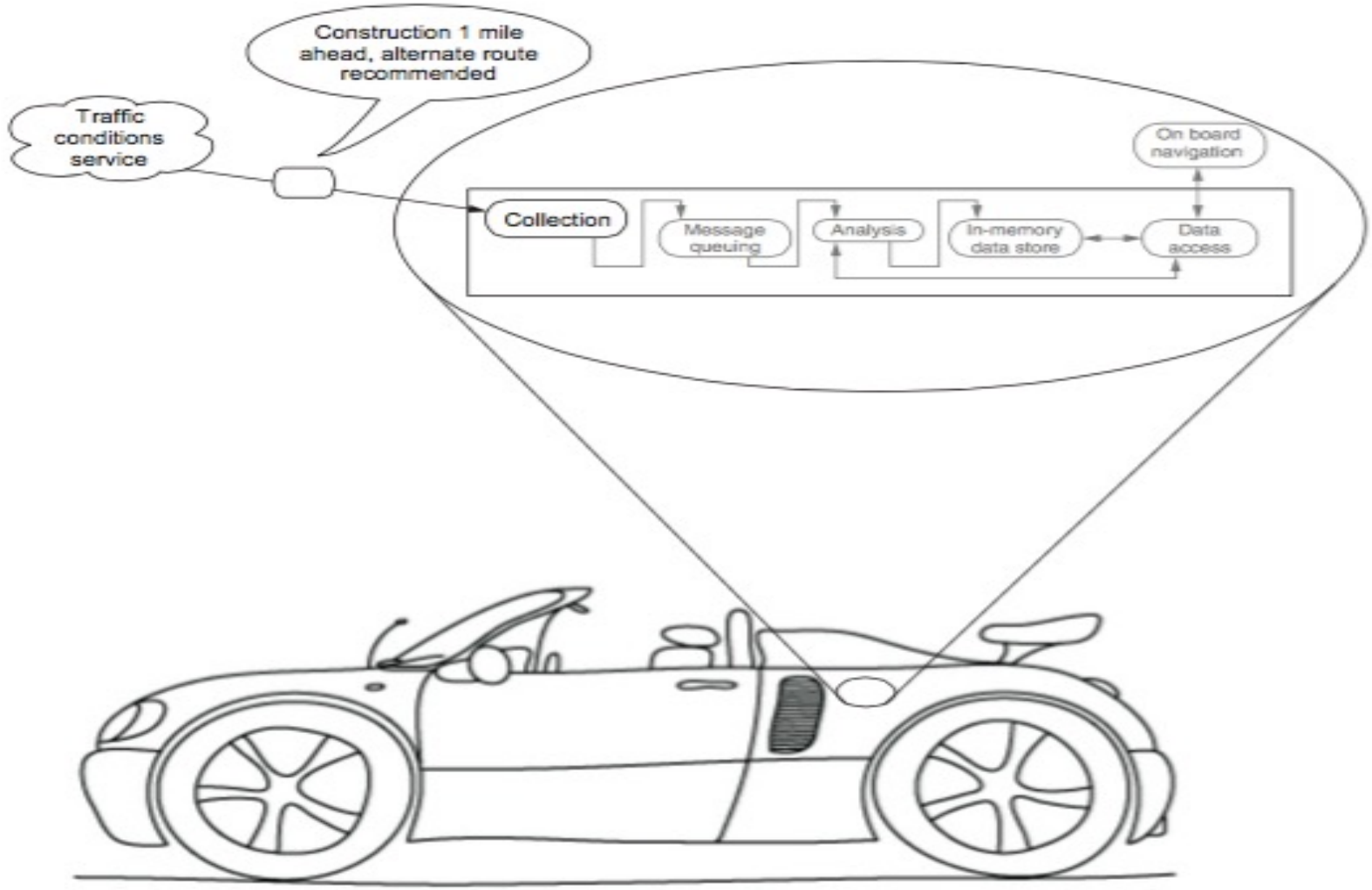
The request and the response happen over the same connection.



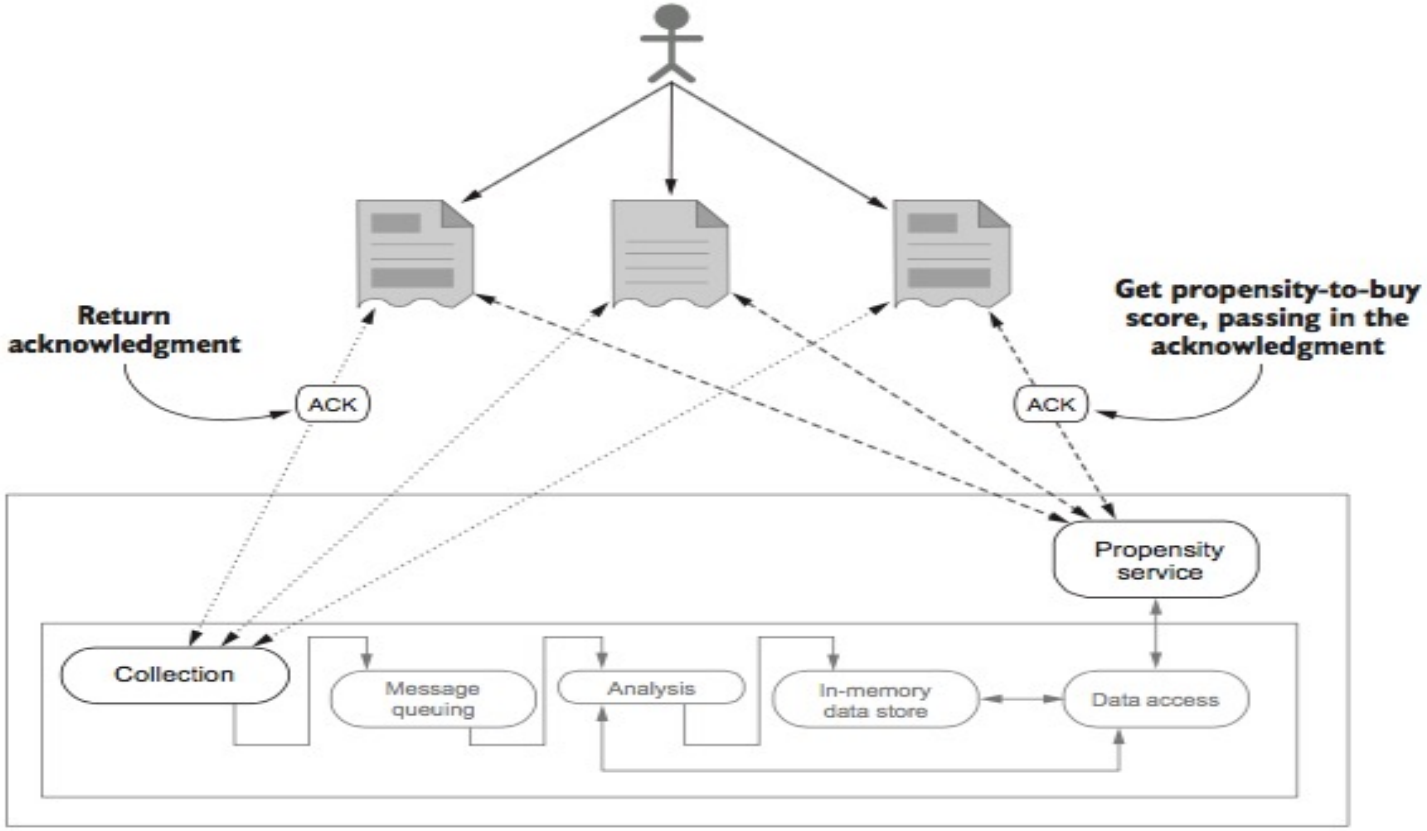
(b) Half-async RR



c) Full-async RR

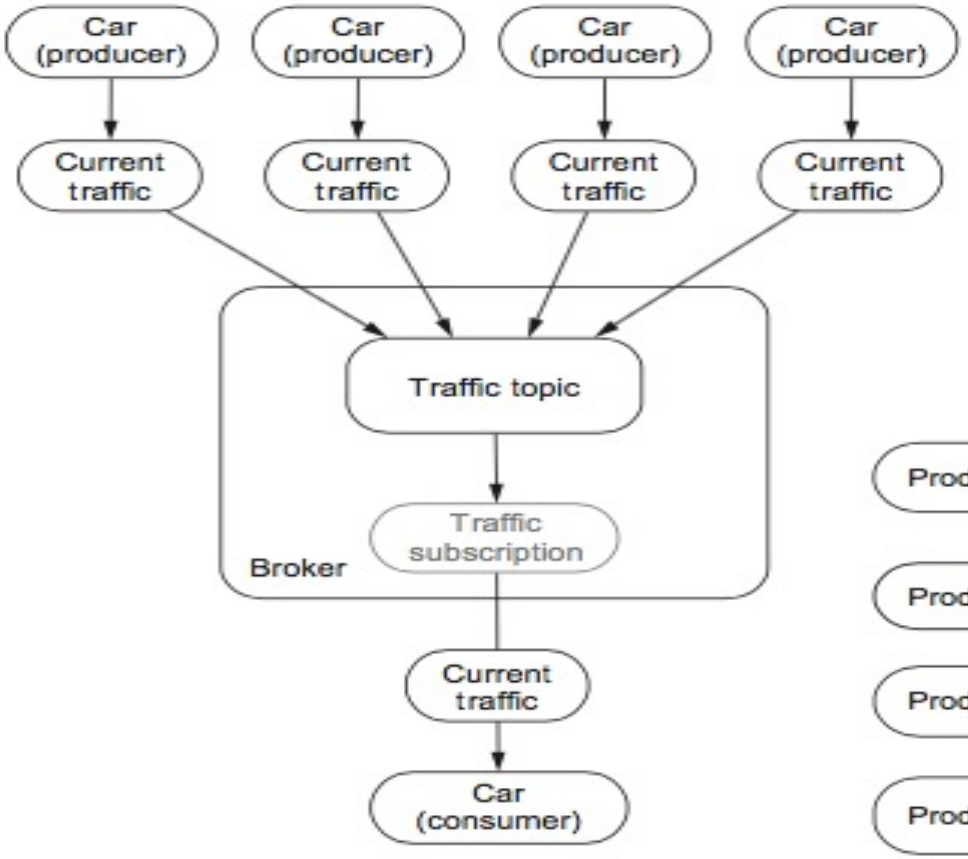

Construction 1 mile ahead, alternate route recommended
Traffic conditions service
On board navigation
Collection
Message queuing
Analysis
In-memory data store
Data access

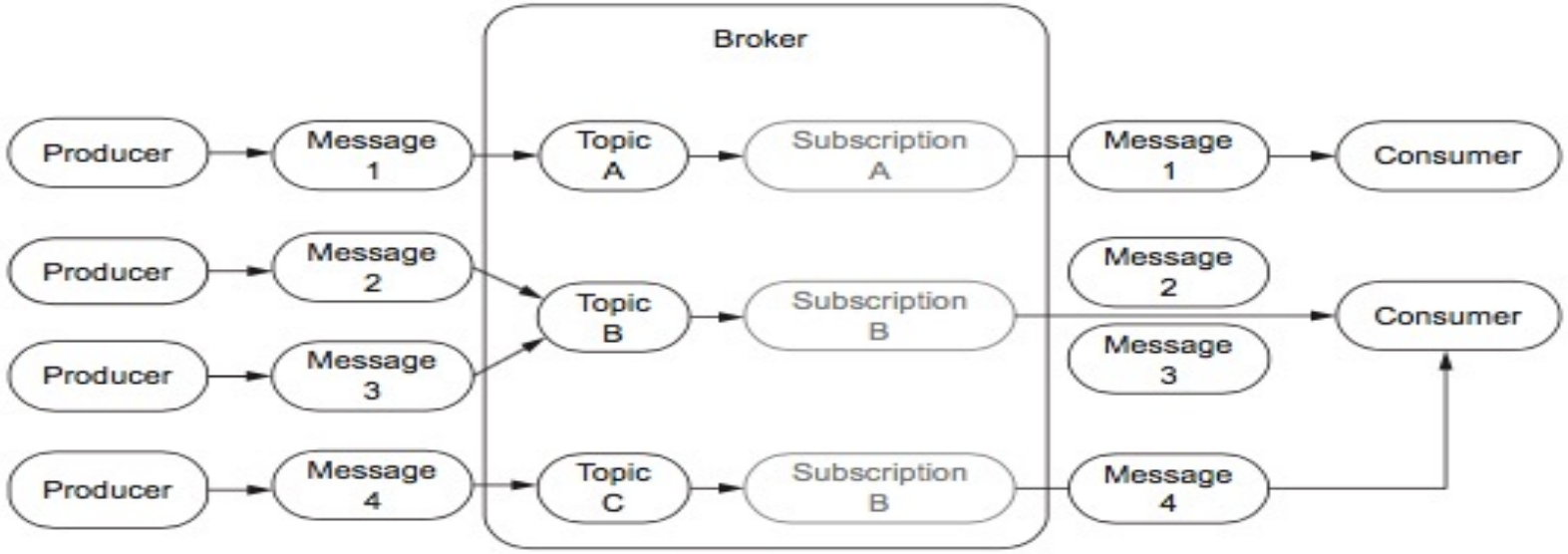
Request/acknowledge



- As the visitor is browsing our site, we're collecting data about each page they visit and every link they click
- The request/acknowledge pattern occurs on the first page they visit
- The acknowledgment is nothing more than a unique identifier
- When we call the propensity service, we can pass the unique identifier we obtained on the very first visit.

Publish/subscribe



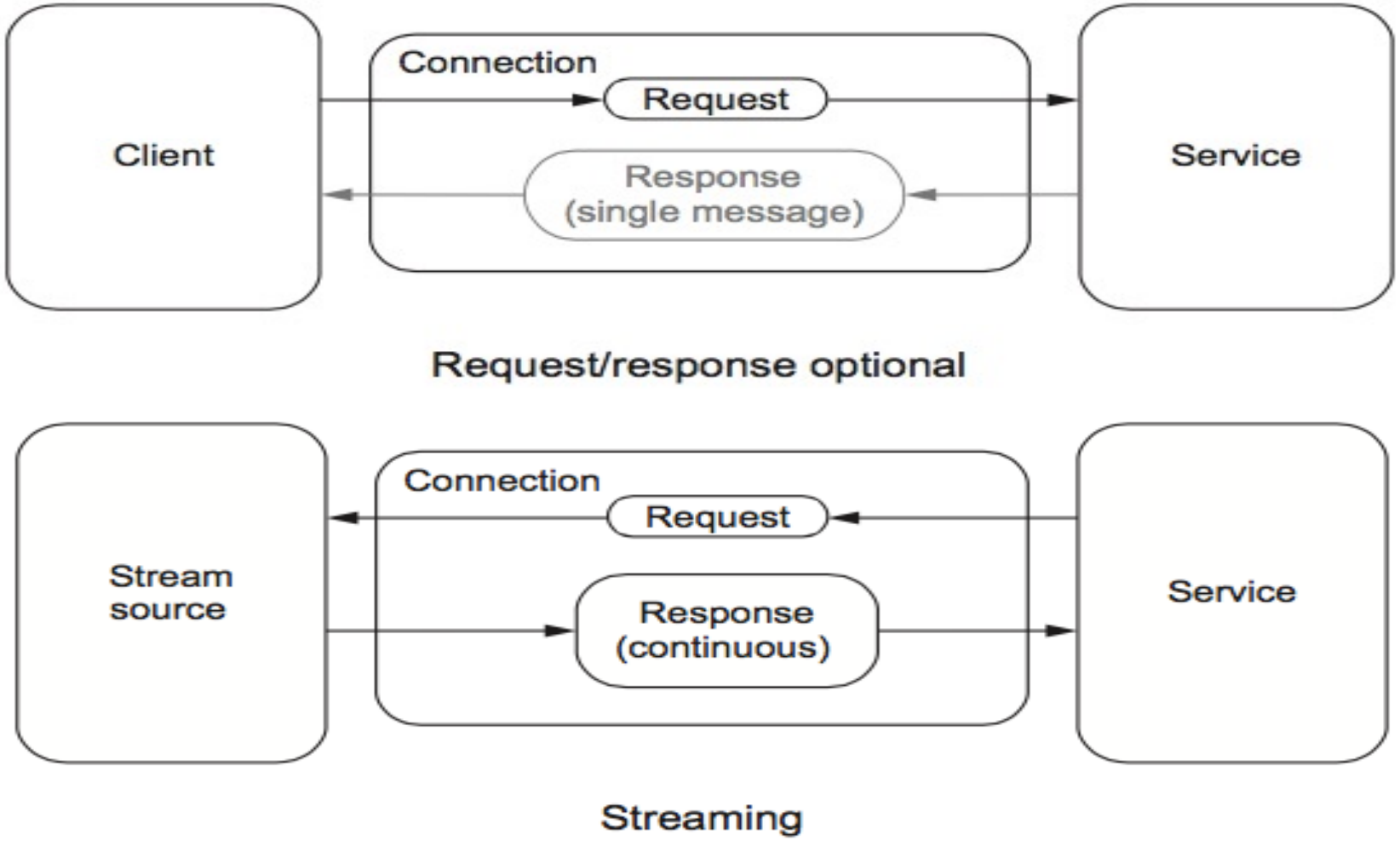


One-way

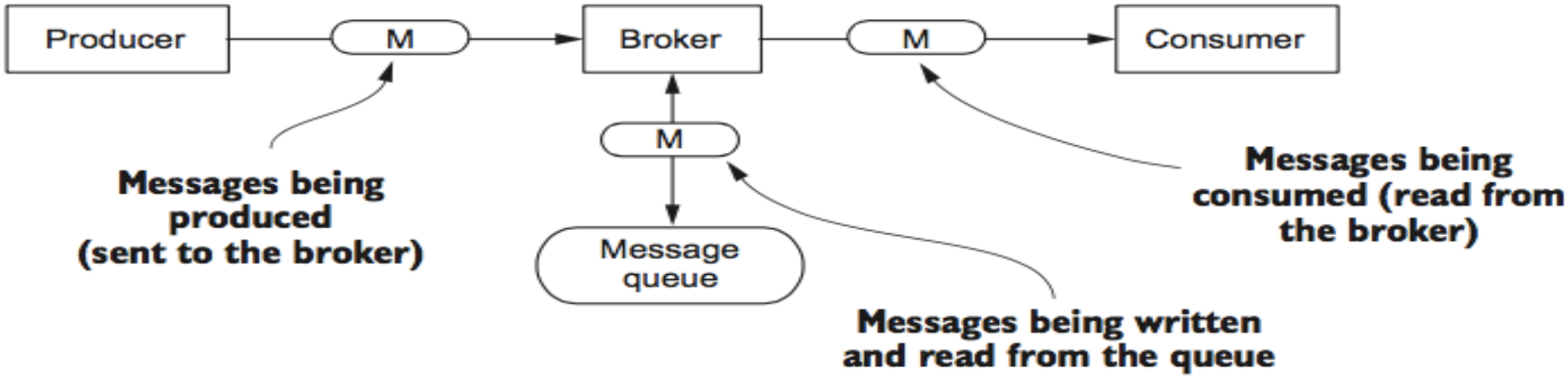
- The “fire and forget” message pattern
- The system making the request doesn’t need a response
- The client does not even know whether the request was received by the service
- Ex:
  - (Environment) sensors
  - Servers send data to the Monitoring System
  - RFID tag to RFID receiver

Stream

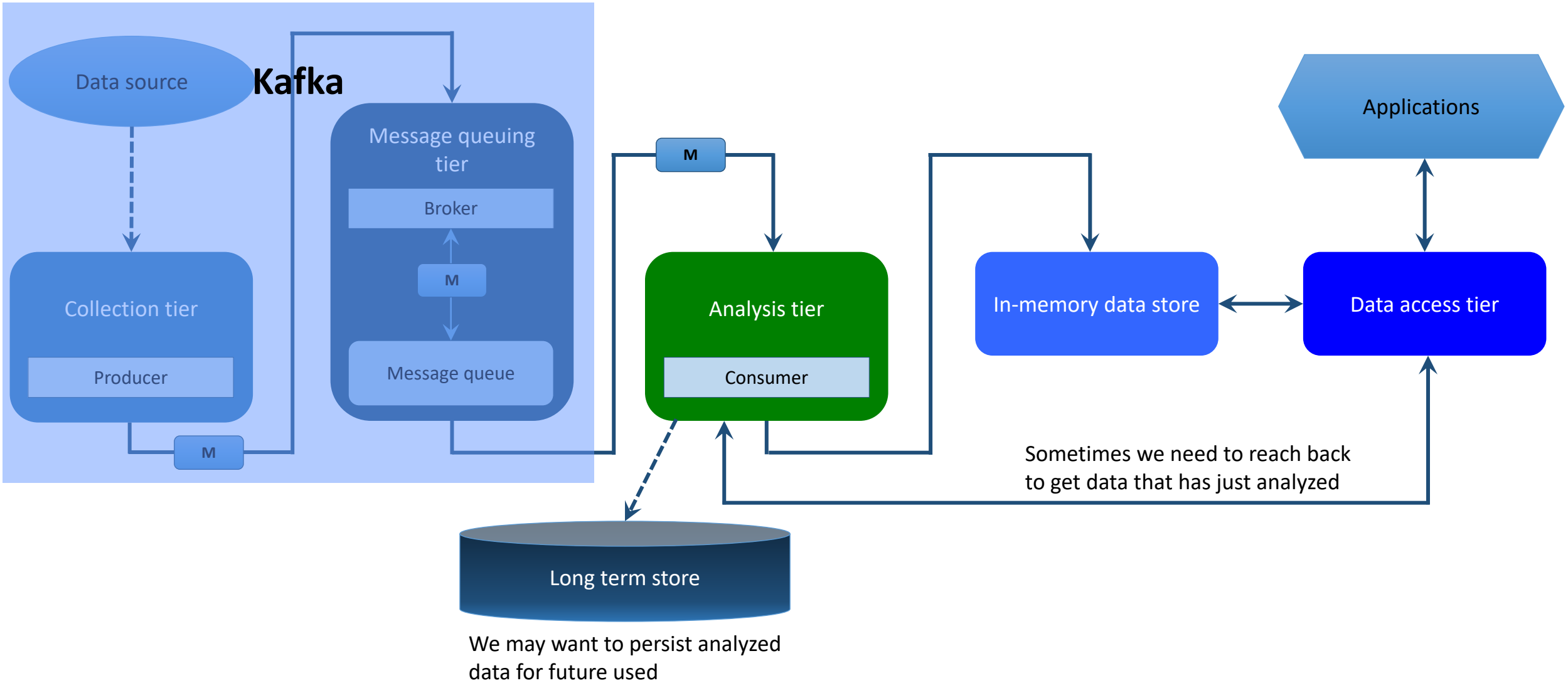
- The stream pattern flips things around as in other patterns
  - the service becomes the client
- A single request results in no data or a continual flow of data as a response
- The collection tier connects to a stream source and pulls data in



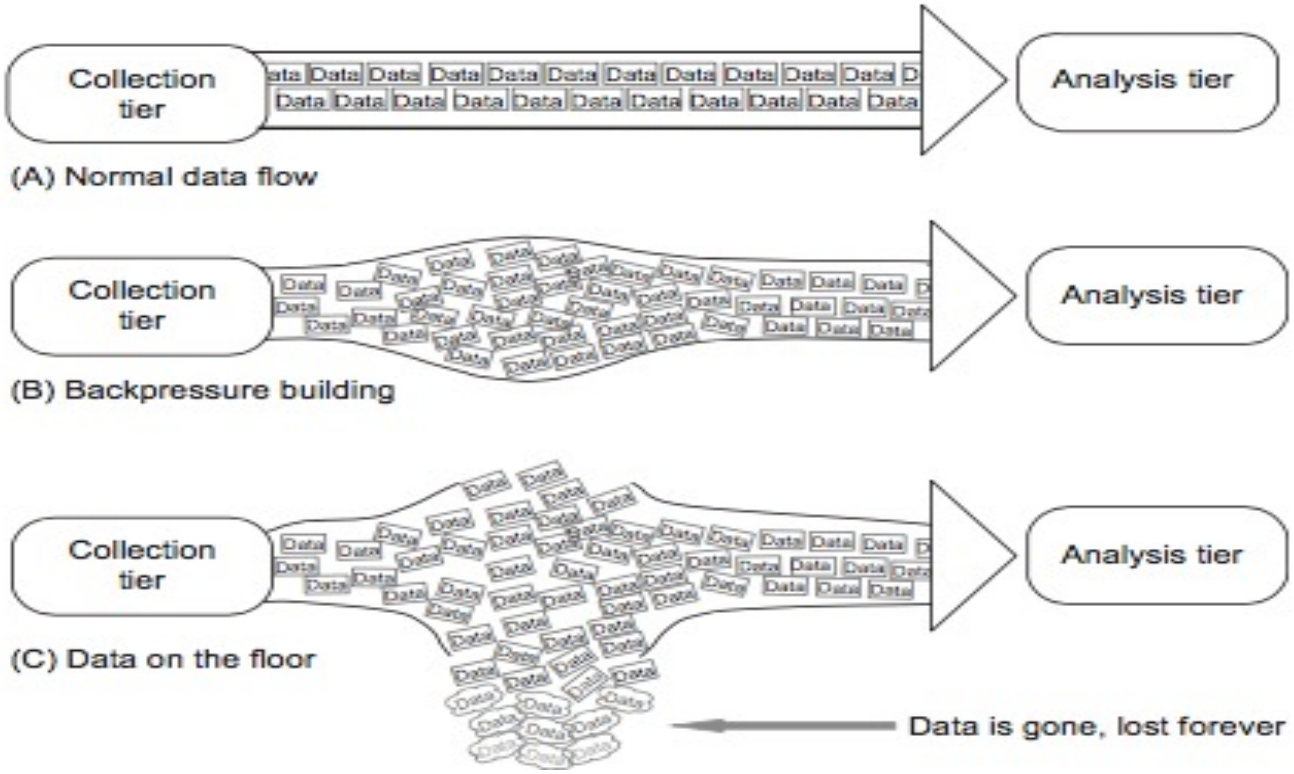
Producer, broker and consumer



- The three core parts to a message queuing system
  - **Producer** sends a message to a broker
  - **Broker** puts the message into a queue
  - **Consumer** reads the message from the broker


Data source
Kafka
Message queuing tier
Broker
M
Message queue
Collection tier
Producer
M
M
Analysis tier
Consumer
In-memory data store
Applications
Data access tier
Sometimes we need to reach back to get data that has just analyzed
Long term store
We may want to persist analyzed data for future used

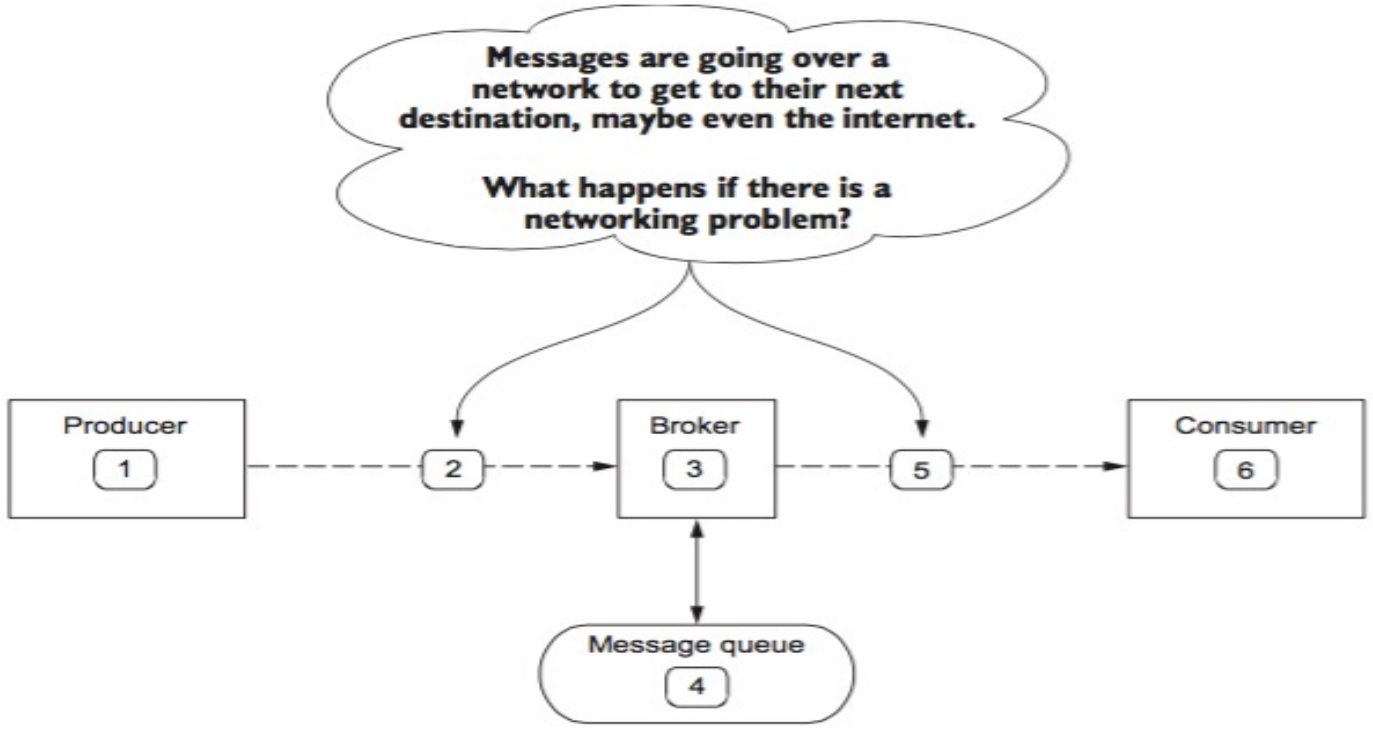
Isolating producers from consumers



- *Step A*: this looks pretty normal and is what we would like to see
- *Step B*: we can tell something is not quite right backpressure is building
- *Step C*: our data pipe broke under pressure, and data is now virtually dropping onto the floor and is gone forever.

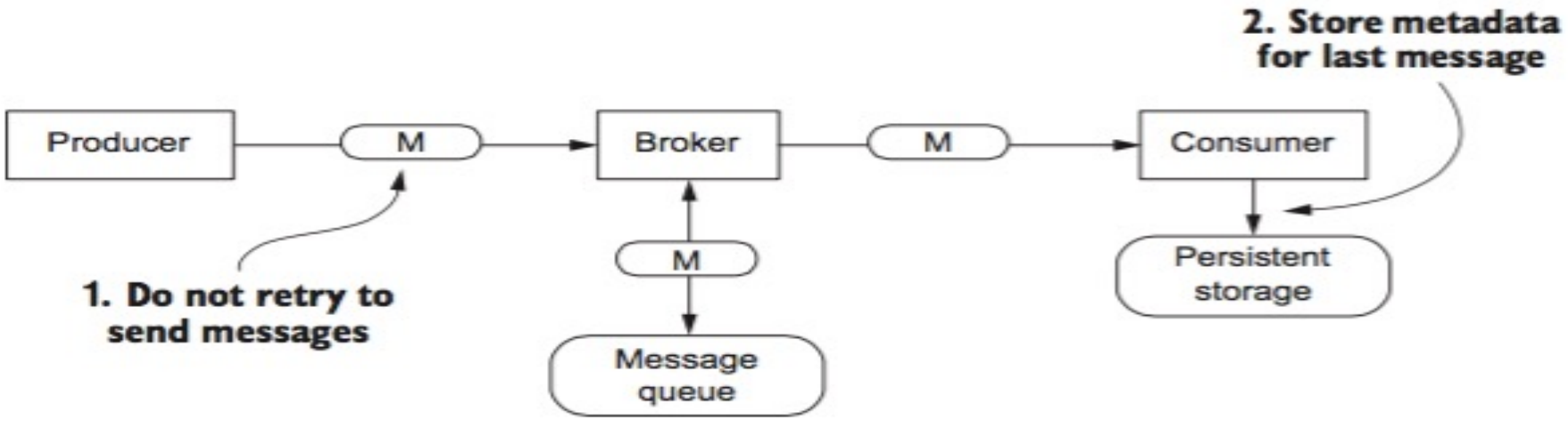
Message delivery semantics

- *At most once*: a message may get lost, but it will never be reread by a consumer
- *At least once*: a message will never be lost, but it may be reread by a consumer
- *Exactly-once*: a message is never lost and is read by a consumer once and only once.



Exactly-once semantics

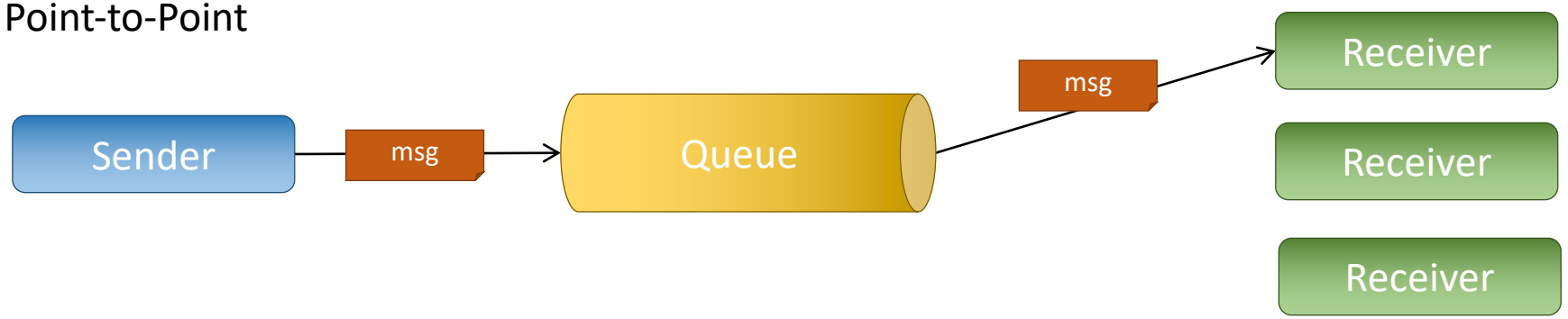
- Apache Kafka and Apache ActiveMQ: NOT support
- Enough metadata about the messages that you can implement exactly-once semantics with some coordination between producer(s) and consumer(s):
  - ➢ *Do not retry to send messages*
    - Read data from the broker to verify that the message you didn't receive an acknowledgment
  - ➢ *Store metadata for last message*
    - Storing some data about the last message we read, e.g. message offset in Apache Kafka
    - Taking into consideration is what to do if there's a failure storing the metadata



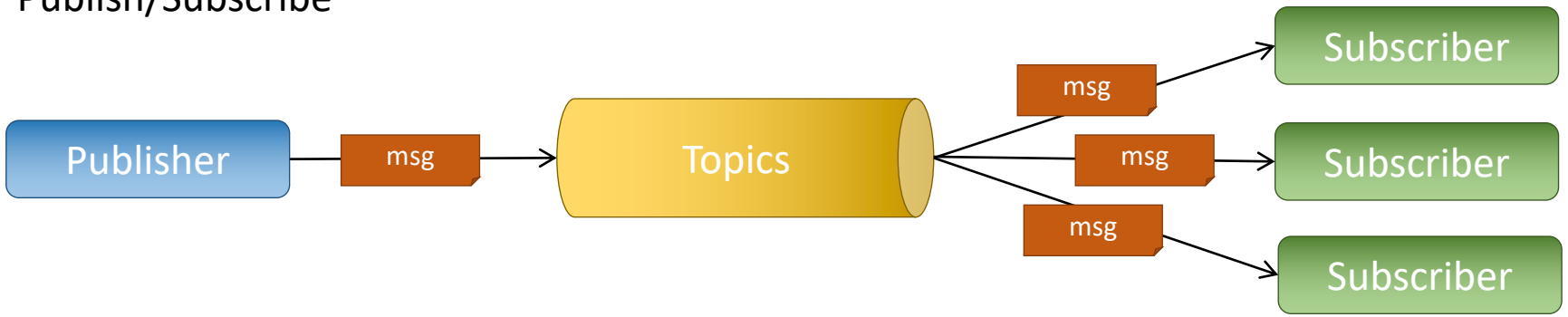
Kafka

Message models

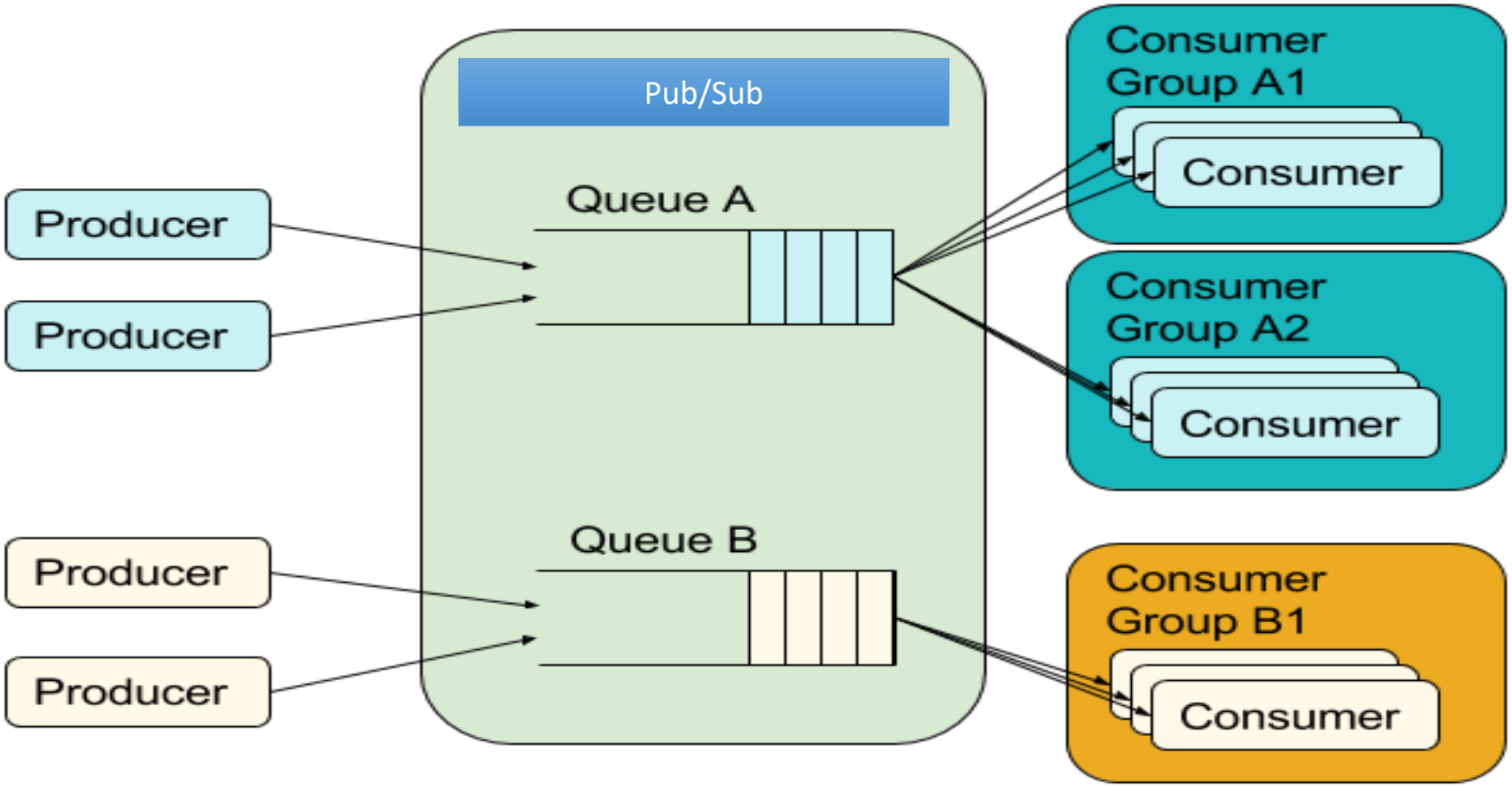
Point-to-Point



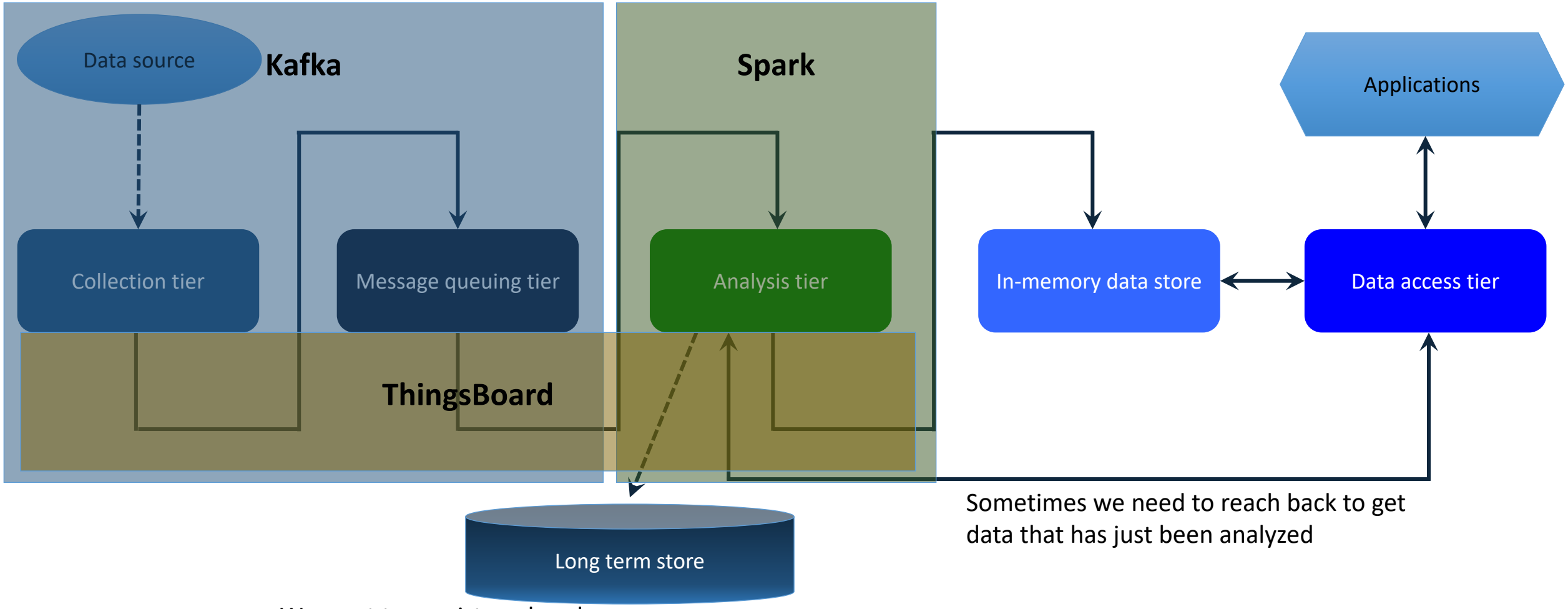
Publish/Subscribe



Publish/Subscribe model



Data streaming architecture: Pub/Sub



Kafka

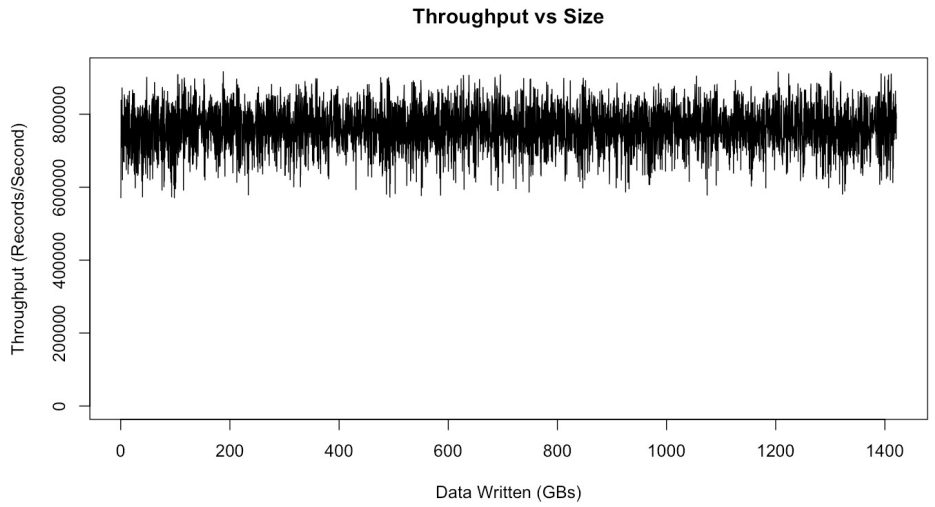
- Kafka is a “publish-subscribe messaging rethought as a distributed commit log”
- Fast
- Scalable
- Durable
- Distributed

Kafka

- Open source project
- Created by LinkedIn, now maintained by Confluence
- Real time, distributed, resilient, fault tolerant and scalable (100s of broker, millions of msg per sec)
- Used by thousands of companies including over 60% of the Fortune 100

How fast is Kafka?

- "Up to 2 million writes/sec on 3 cheap machines"
  - Using 3 producers on 3 different machines, 3x async replication
    - Only 1 producer/machine because NIC already saturated
- Sustained throughput as stored data grows
  - Slightly different test config than 2M writes/sec above.

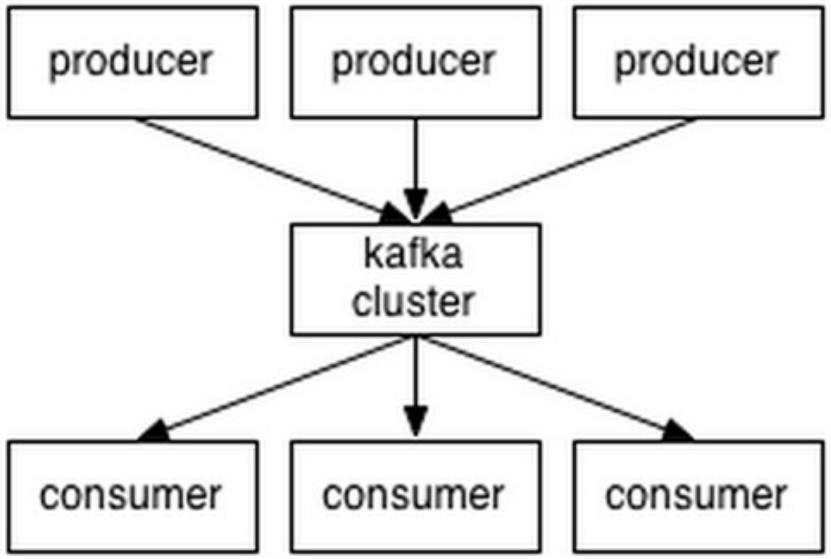


Why is Kafka so fast?

- Fast **writes**:
  - While Kafka persists all data to disk, essentially all writes go to the **page cache** of OS, i.e. RAM
- Fast **reads**:
  - Very efficient to transfer data from page cache to a network **socket**
  - Linux: **sendfile()** system call
- Combination of the two = fast Kafka!
  - Example (Operations): On a Kafka cluster where the consumers are mostly caught up you will see no read activity on the disks as they will be serving data entirely from cache

A first look

- The who is who
  - **Producers** write data to **brokers**
  - **Consumers** read data from **brokers**
  - All this is distributed
- The data
  - Data is stored in **topics**
  - **Topics** are split into **partitions**, which are **replicated**



Apache Kafka

Topic

Offset

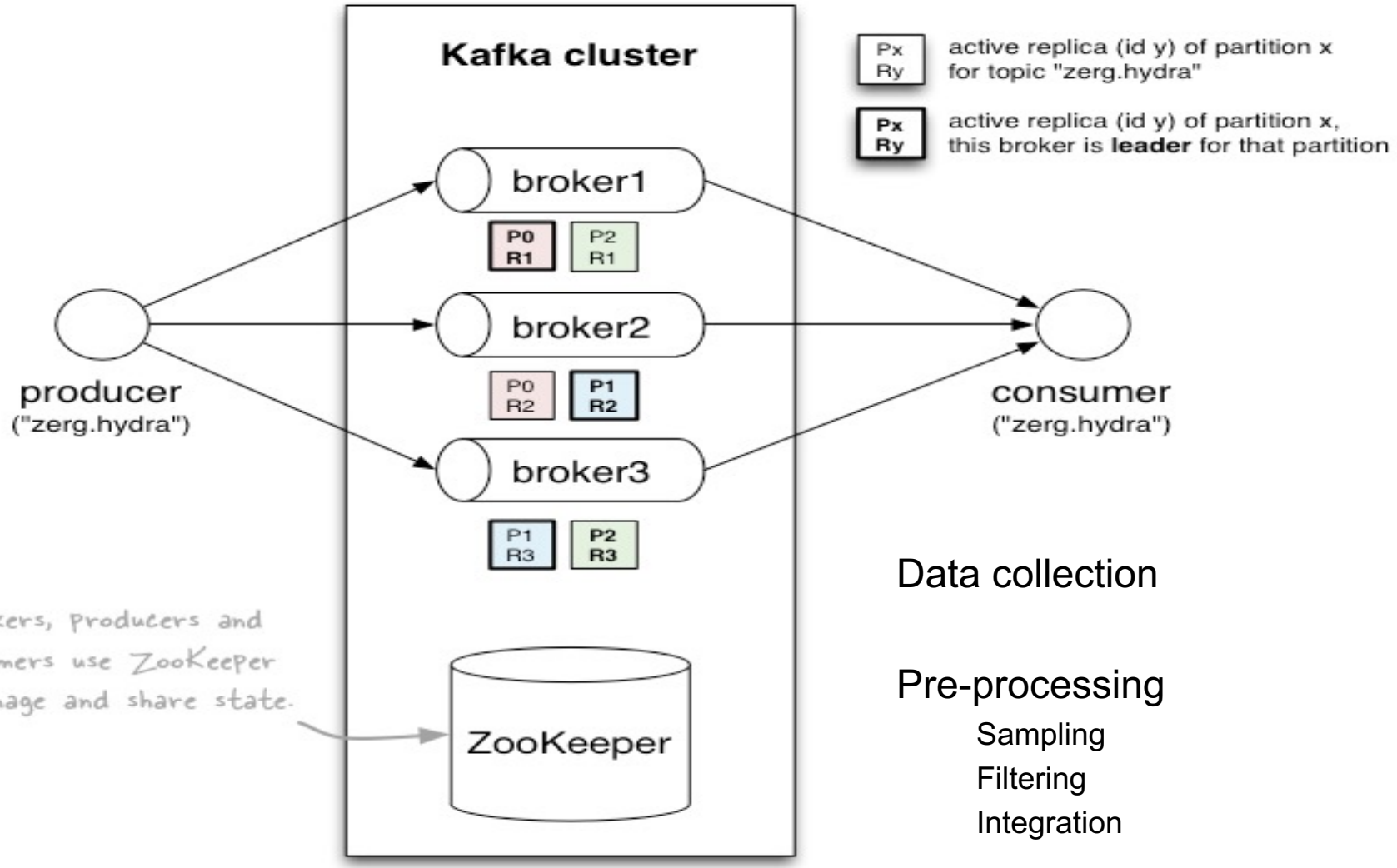
Broker

Partition

Replica

Kafka cluster

ZooKeeper



Data collection

Pre-processing

- Sampling
- Filtering
- Integration

Topic, Partition and Offset (1)

Topics: Place to store/read all messages

- Each messages must be organized into at least one specific topic
- Each topic has its own name as id

Topics are divided into a number of **partitions**

- Partition contains records in an immutable order
- Each record is identified using a non-stop incremental integer number called **offset**
- Allows multiple consumers to read from a topic in parallel

Topic, Partition and Offset (2)

Be careful!:

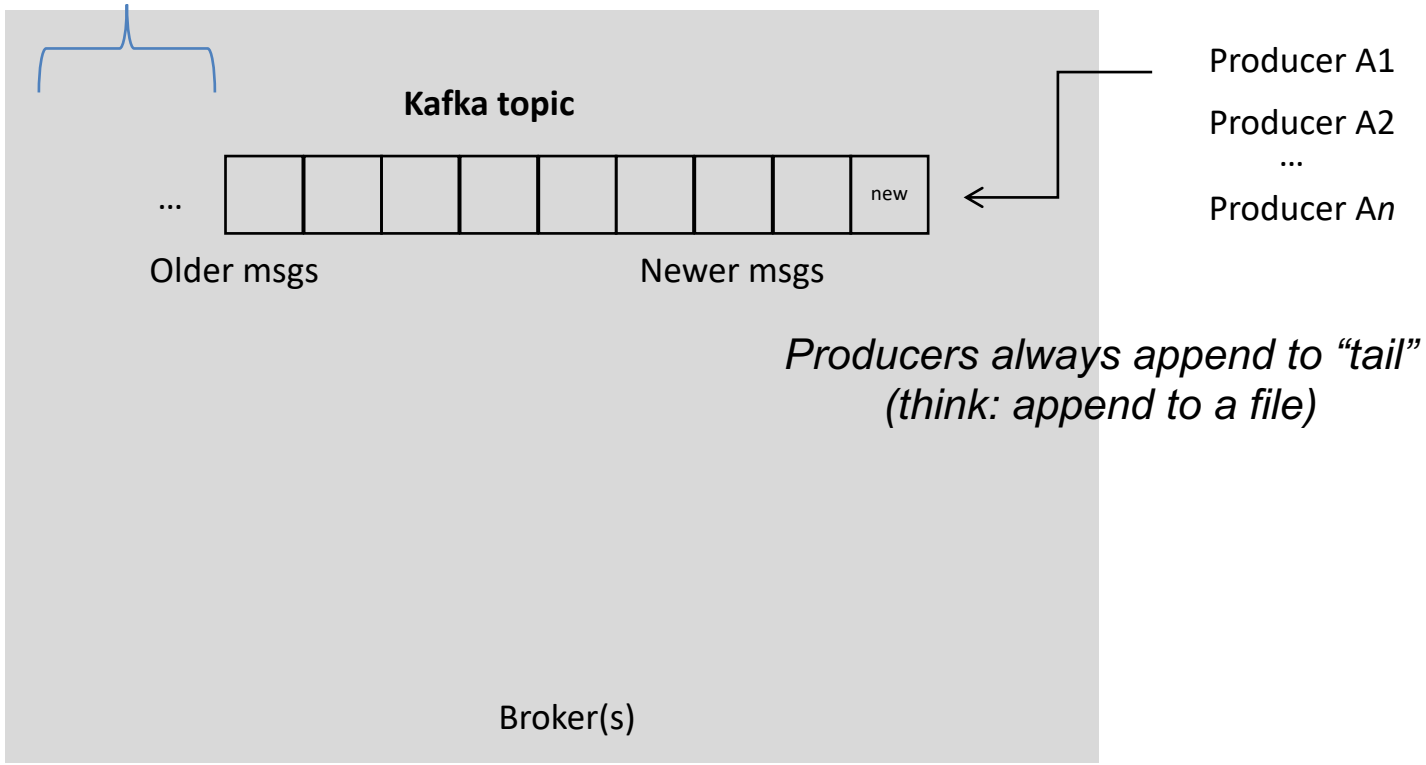
- Offset comparison and record order is only guaranteed within a partition
- Record has a limited lifetime (default: 1 week)
- Once the data is written, you cannot update it (but you can delete).
- Incoming data is stored at a random partition unless a key is provided

Topic (1)

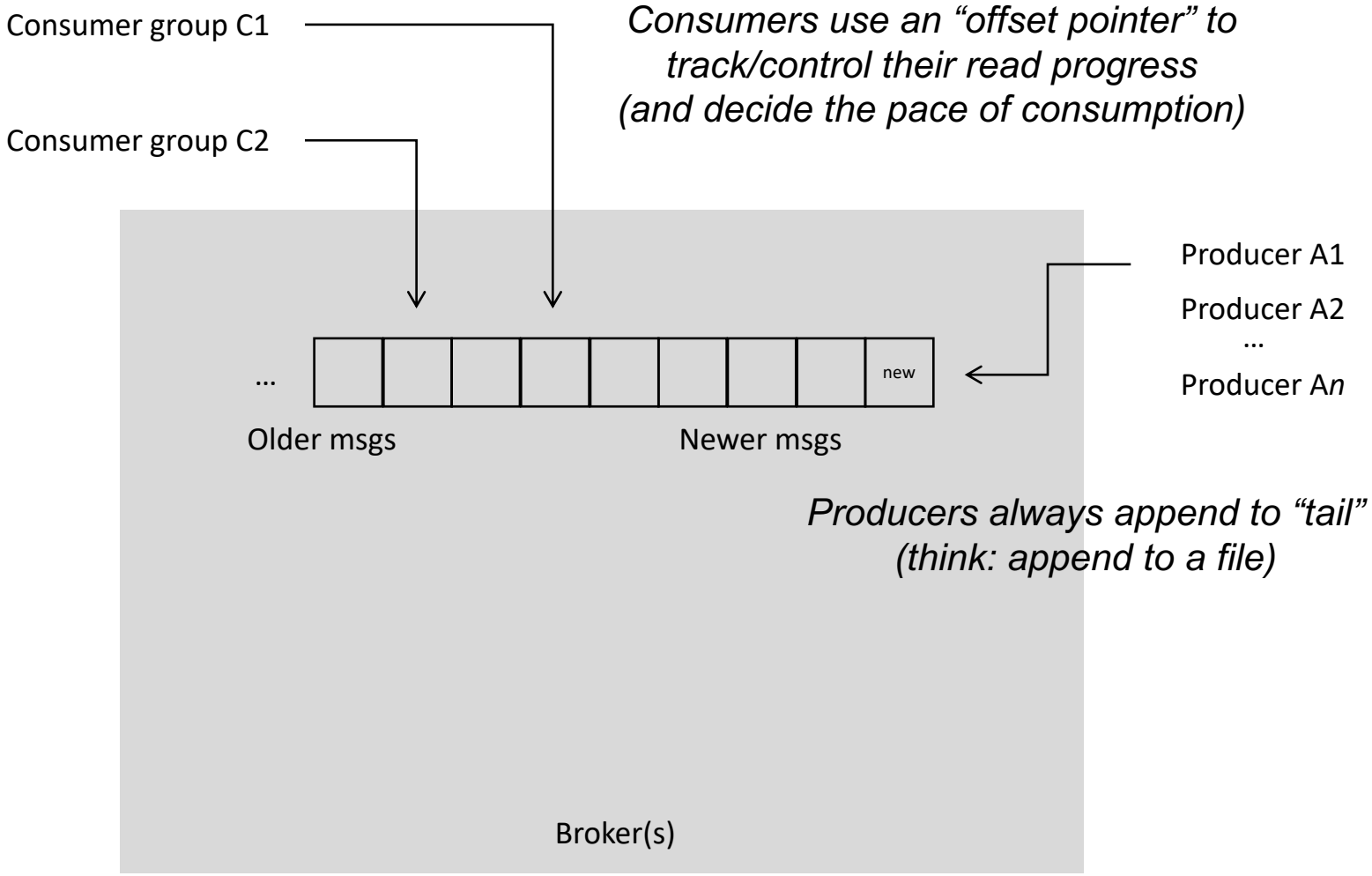
Topic: feed name to which messages are published

Example: “zerg.hydra”

Kafka prunes “head” based on age or max size or “key”

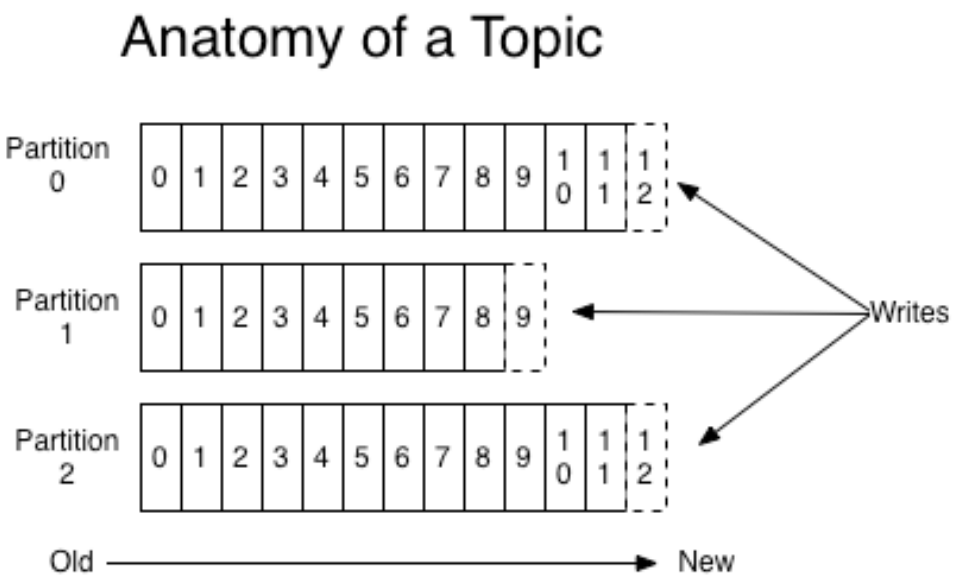


Topic (2)


Consumer group C1
Consumer group C2
Consumers use an “offset pointer” to track/control their read progress (and decide the pace of consumption)
Producer A1
Producer A2
…
Producer An
…
new
Older msgs
Newer msgs
Producers always append to “tail” (think: append to a file)
Broker(s)

Partitions (1)

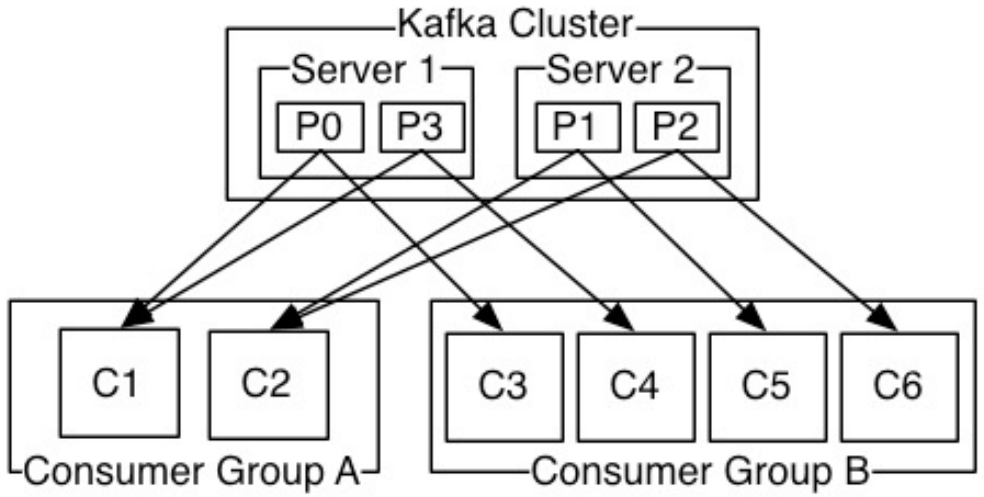
- A topic consists of **partitions**
- Partition: **ordered + immutable** sequence of messages that is continually appended to



Partitions (2)

- No. of partitions of a topic is configurable
- No. of partitions determines **max** consumer (group) parallelism

    cf. parallelism of Storm's KafkaSpout via builder.setSpout(,,*N*)

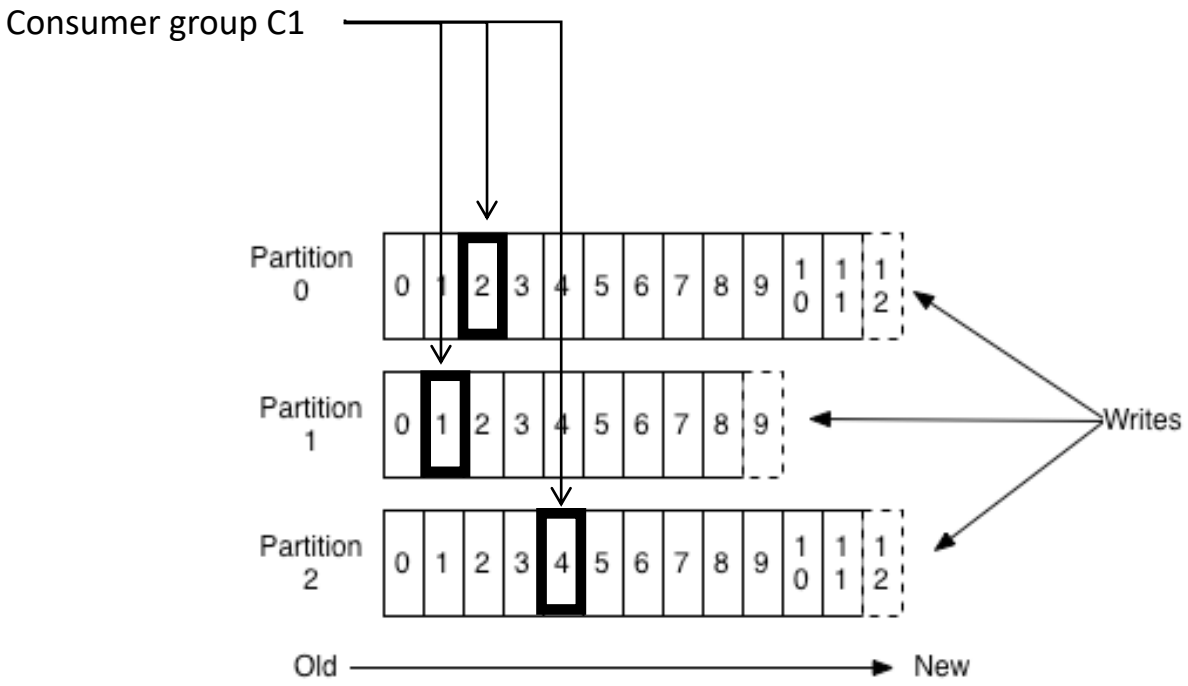


Consumer group A, with 2 consumers, reads from a 4-partition topic

Consumer group B, with 4 consumers, reads from the same topic

Partition offsets

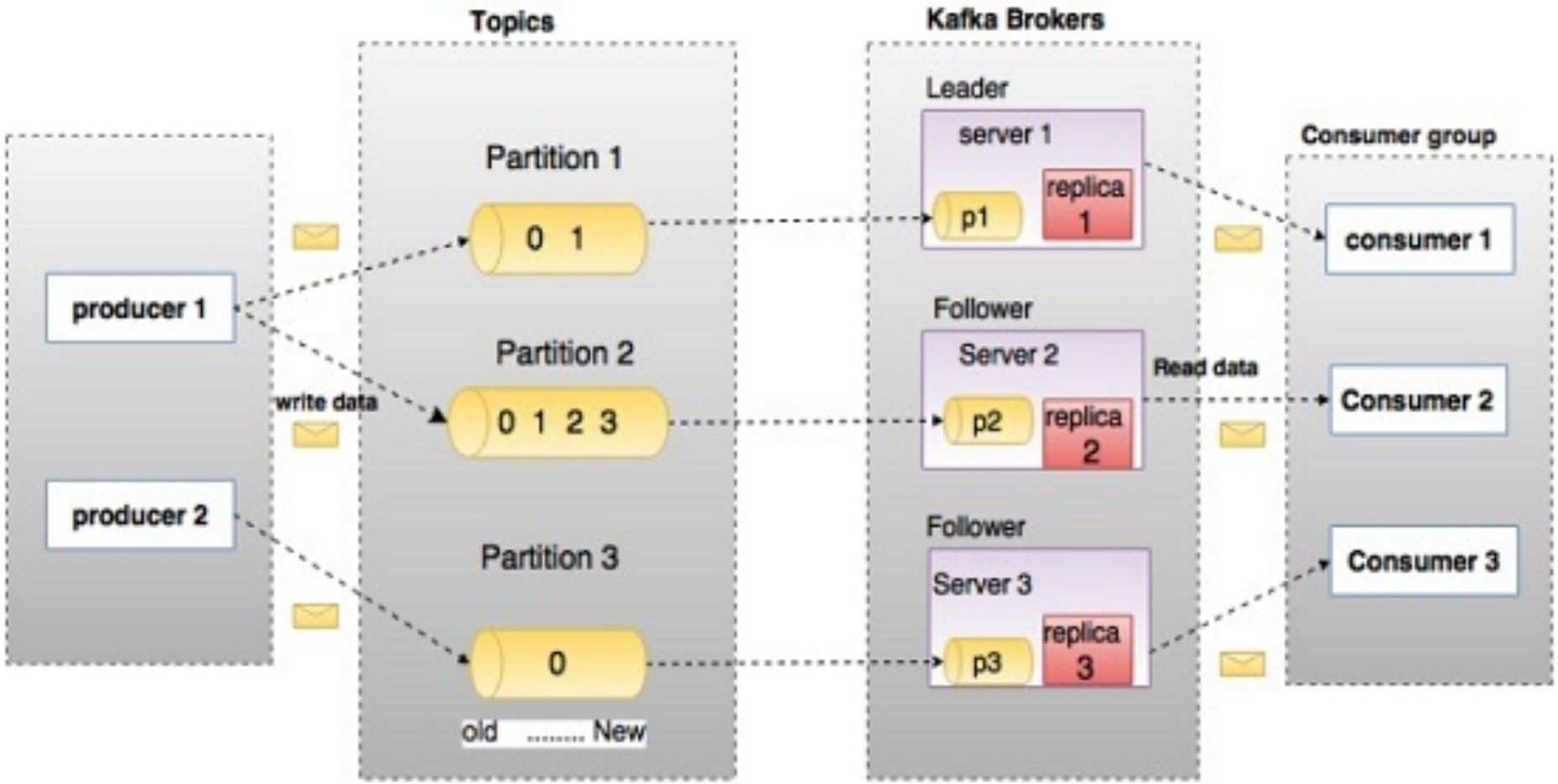
- **Offset**: messages in the partitions are each assigned a unique (per partition) and sequential id called the *offset*
  - Consumers track their pointers via *(offset, partition, topic)* tuples



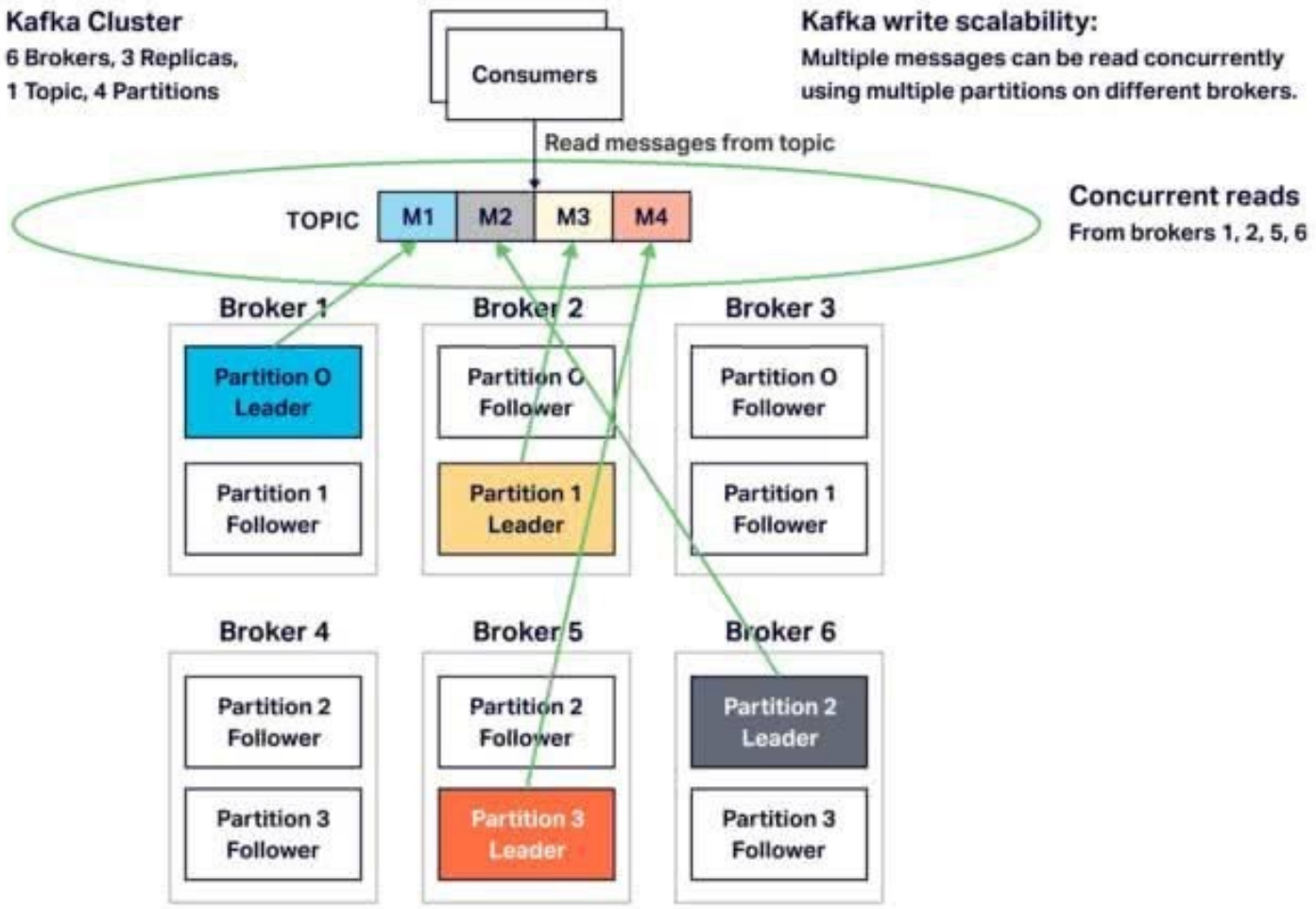
Replications of a partition

- **Replicas:** “backups” of a partition
  - They exist solely to prevent data loss
  - Replicas are never read from, never written to
    - They do NOT help to increase producer or consumer parallelism!
- Kafka tolerates *(numReplicas - 1)* dead brokers before losing data
  - LinkedIn: numReplicas = 2 → 1 broker can die

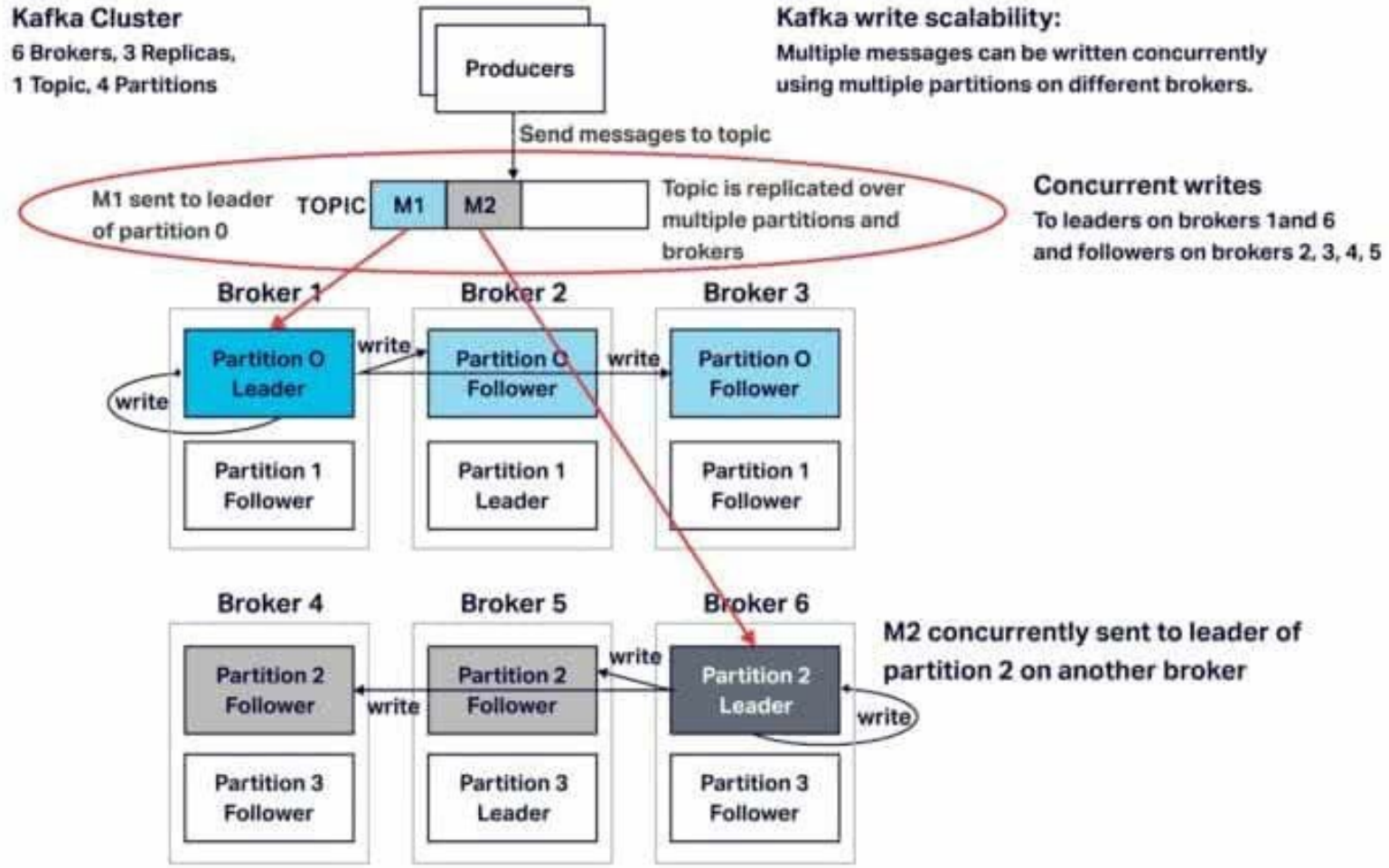
Replication



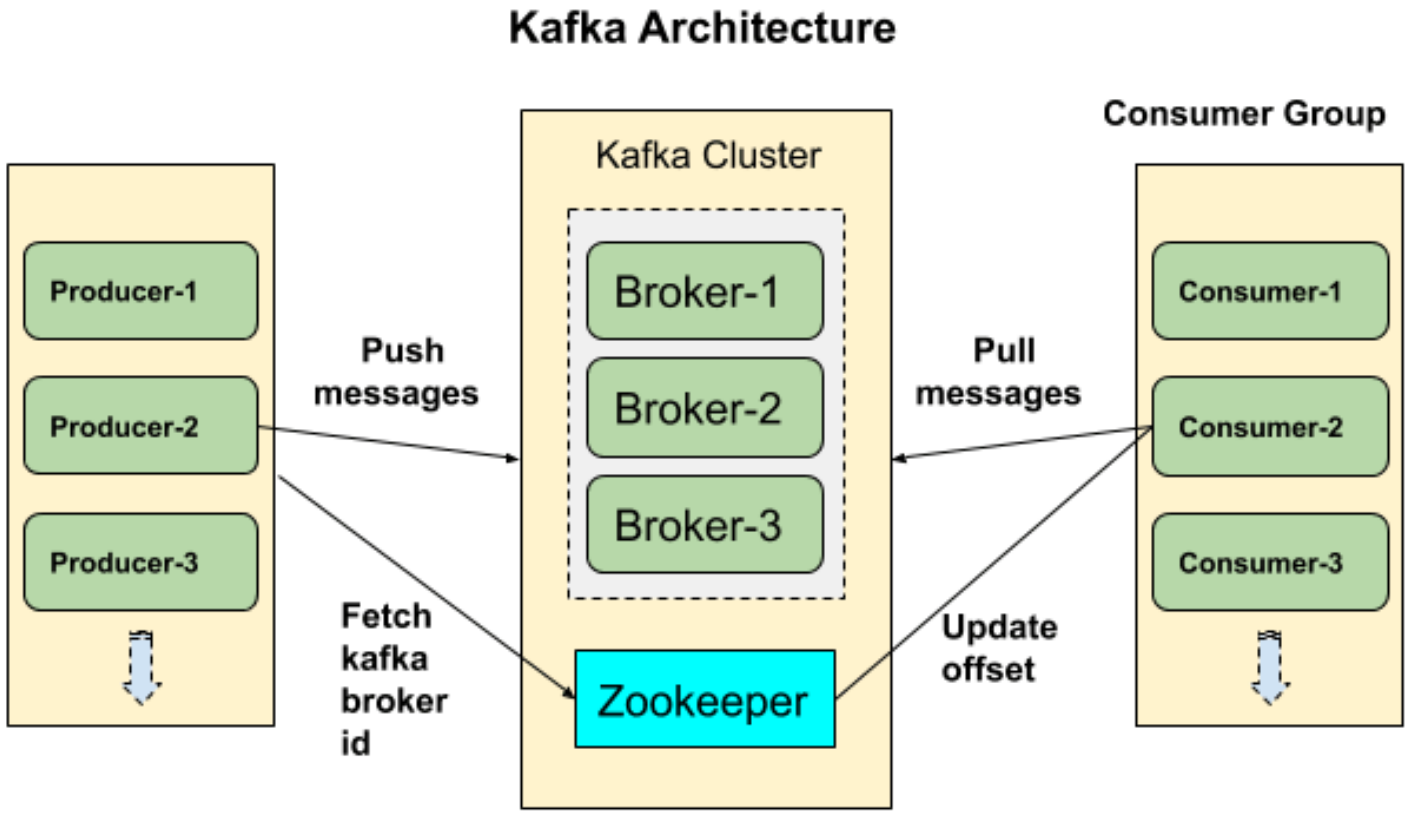
Partition


Kafka Cluster
6 Brokers, 3 Replicas,
1 Topic, 4 Partitions
Consumers
Kafka write scalability:
Multiple messages can be read concurrently
using multiple partitions on different brokers.
Read messages from topic
TOPIC
M1
M2
M3
M4
Concurrent reads
From brokers 1, 2, 5, 6
Broker 1
Broker 2
Broker 3
Partition 0
Leader
Partition 0
Follower
Partition 0
Follower
Partition 1
Follower
Partition 1
Leader
Partition 1
Follower
Broker 4
Broker 5
Broker 6
Partition 2
Follower
Partition 2
Follower
Partition 2
Leader
Partition 3
Follower
Partition 3
Leader
Partition 3
Follower

Write


Kafka Cluster
6 Brokers, 3 Replicas,
1 Topic, 4 Partitions
Producers
Kafka write scalability:
Multiple messages can be written concurrently
using multiple partitions on different brokers.
Send messages to topic
M1 sent to leader
of partition 0
TOPIC
M1
M2
Topic is replicated over
multiple partitions and
brokers
Concurrent writes
To leaders on brokers 1and 6
and followers on brokers 2, 3, 4, 5
Broker 1
Broker 2
Broker 3
Partition 0
Leader
write
Partition 0
Follower
write
Partition 0
Follower
write
Partition 1
Follower
Partition 1
Leader
Partition 1
Follower
Broker 4
Broker 5
Broker 6
Partition 2
Follower
write
Partition 2
Follower
write
Partition 2
Leader
write
M2 concurrently sent to leader of
partition 2 on another broker
Partition 3
Follower
Partition 3
Leader
Partition 3
Follower

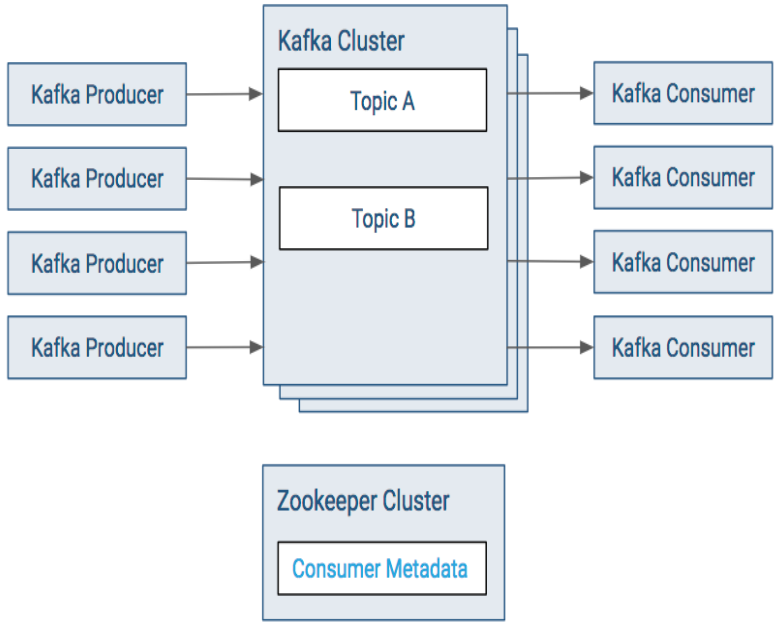
ZooKeeper (1)


Kafka Architecture
Kafka Cluster
Consumer Group
Producer-1
Producer-2
Producer-3
Push messages
Broker-1
Broker-2
Broker-3
Pull messages
Consumer-1
Consumer-2
Consumer-3
Fetch kafka broker id
Zookeeper
Update offset

ZooKeeper (2)

Zookeeper is a centralized service to maintain naming and configuration data, provide flexible and robust synchronization within distributed systems

- Zookeeper keeps track of status of the Kafka brokers, topics, partitions,... and notify all changes to Kafka
- **Kafka cannot run without Zookeeper!**



ZooKeeper (3)

How Zookeeper is helping Kafka:

- Kafka Brokers' state & quotas: Zookeeper determines the state of all brokers of the cluster, following the replication option. It also keeps track of how much data is each client allowed to read and write
- Configuration of Topics: Zookeeper keeps all configuration regarding all the topics including the list of existing topics, the number of partitions for each topic, the location of all the replicas, list of configuration overrides for all topics, and which node is the preferred leader, etc.
- Access Control Lists: Access control lists or ACLs for all the topics are also maintained within Zookeeper
- Cluster membership: Zookeeper also maintains a list of all the brokers that are functioning at any given moment and are a part of the cluster
- Controller Election: If a node for some reason is shutting down, Zookeeper will select one of working replicas to act as partition leaders
- Consumer Offsets and Registry: Zookeeper keeps all information about how many messages Kafka consumer consumes

Workflow of Pub/Sub messaging (2)

- Once Kafka receives the messages from producers, it forwards these messages to the consumers
- Consumer will receive the message and process it
- Once the messages are processed, consumer will send an acknowledgement to the Kafka broker
- Once Kafka receives an acknowledgement, it changes the offset to the new value and updates it in the Zookeeper. Since offsets are maintained in the Zookeeper, the consumer can read next message correctly even during server outrages
- This above flow will repeat until the consumer stops the request
- Consumer has the option to rewind/skip to the desired offset of a topic at any time and read all the subsequent messages.

Workflow of consumer group (1)

- Producers send message to a topic in a regular interval
- Kafka stores all messages in the partitions configured for that particular topic similar to the earlier scenario
- A single consumer subscribes to a specific topic, assume Topic-01 with Group ID as Group-1
- Kafka interacts with the consumer in the same way as Pub-Sub Messaging until new consumer subscribes the same topic, Topic-01 with the same Group ID as Group-1
- Once the new consumer arrives, Kafka switches its operation to share mode and shares the data between the two consumers. This sharing will go on until the number of consumers reach the number of partition configured for that particular topic.

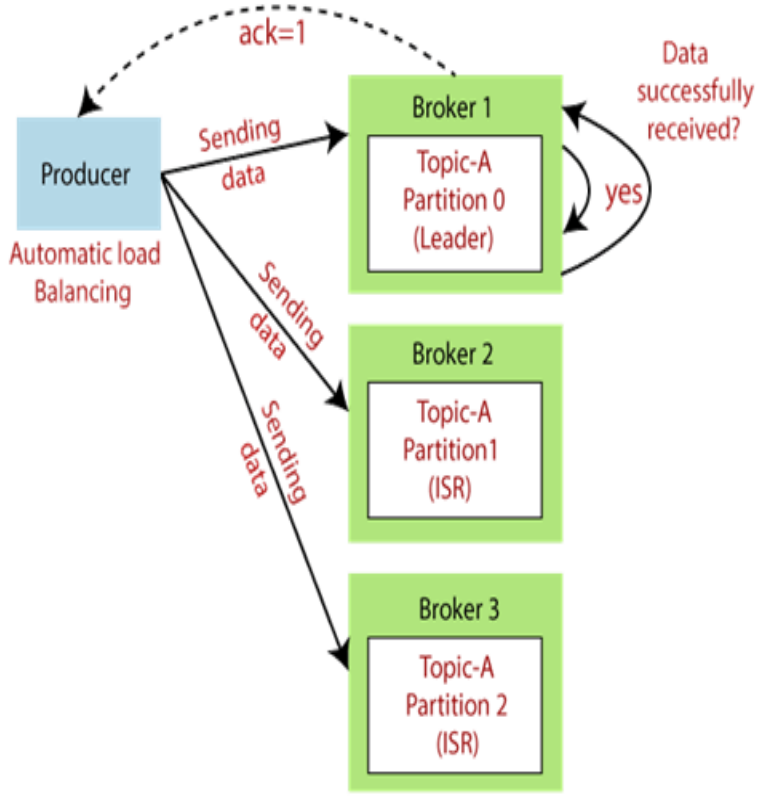
Workflow of consumer group (2)

- Once the number of consumer exceeds the number of partitions, the new consumer will not receive any further message until any one of the existing consumer unsubscribes. This scenario arises because each consumer in Kafka will be assigned a minimum of one partition and once all the partitions are assigned to the existing consumers, the new consumers will have to wait
- This feature is also called as Consumer Group. In the same way, Kafka will provide the best of both the systems in a very simple and efficient manner.

Producer and Consumer (1)

Producers send data to broker to write into topics

- Producer is provided which broker and partition to store its data



Receiving Successfull data acknowledgement

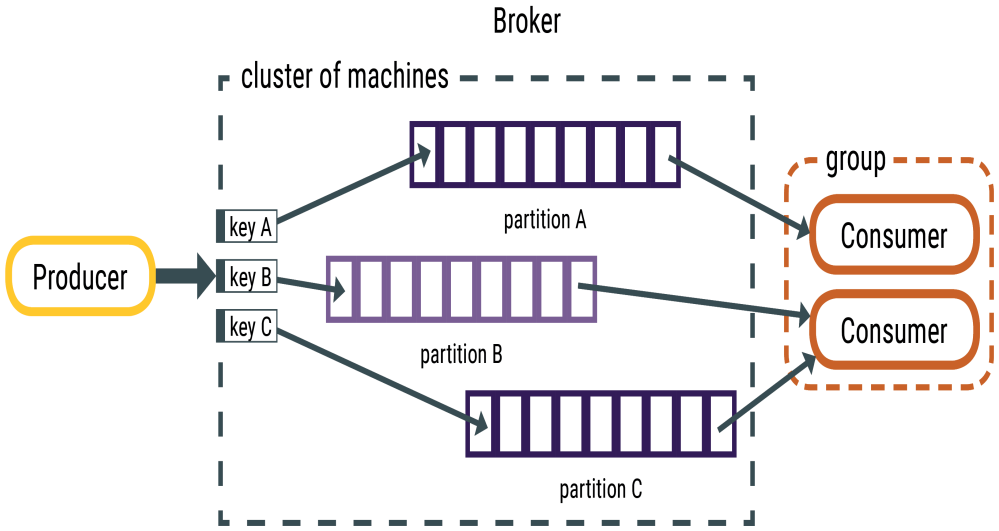
To guarantee the data write operation, we can set the producer wait for the **data acknowledgement,** based on 3 options:

- **ack=0:** Producer do not need to know about any ack
- **ack=1**: Producer will wait for the leader ack
- **ack=2**: Producer will wait for the acks from leader and replicas

Producer and Consumer (2)

Producer can use a **message key** while sending data

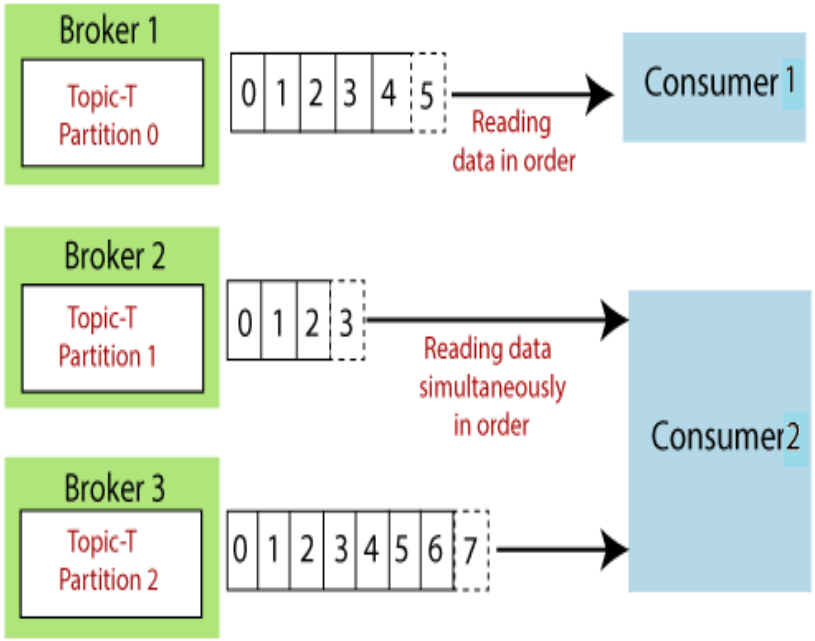
- If a key is provided, all message with a same key will be stored at a same partition
- Otherwise, the data is written round robin at brokers
- Use message key to make sure your data is ordered for a specific field (ex: *user_id*)



Producer and Consumer (3)

Consumer retrieve data of a topic from brokers

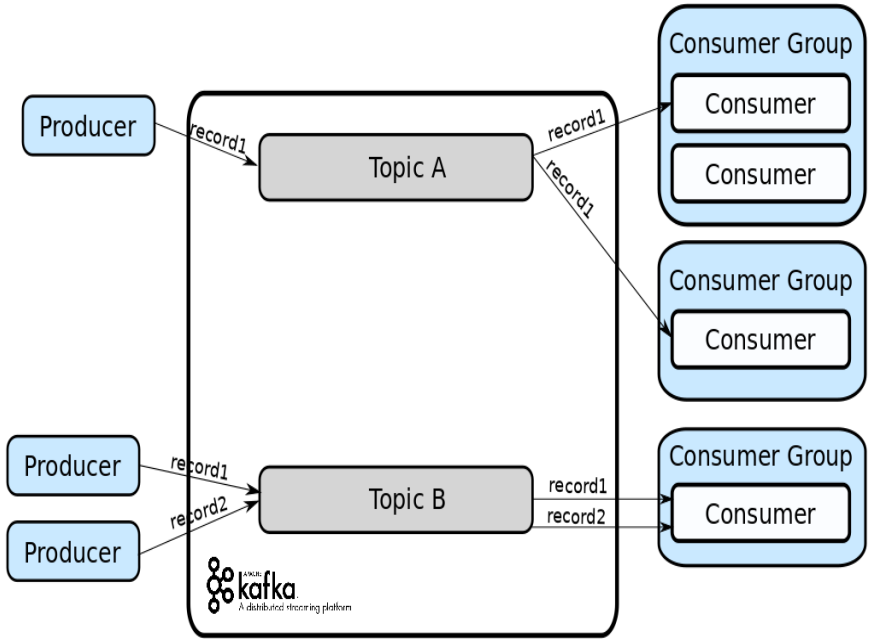
- Consumer is also provided which broker and partition to **pull** data
- Record is read in order within each partition
- Consumers can read messages starting from any offset point they choose.



Producer and Consumer (4)

Consumers are organized into **Consumer Group** (using group name)

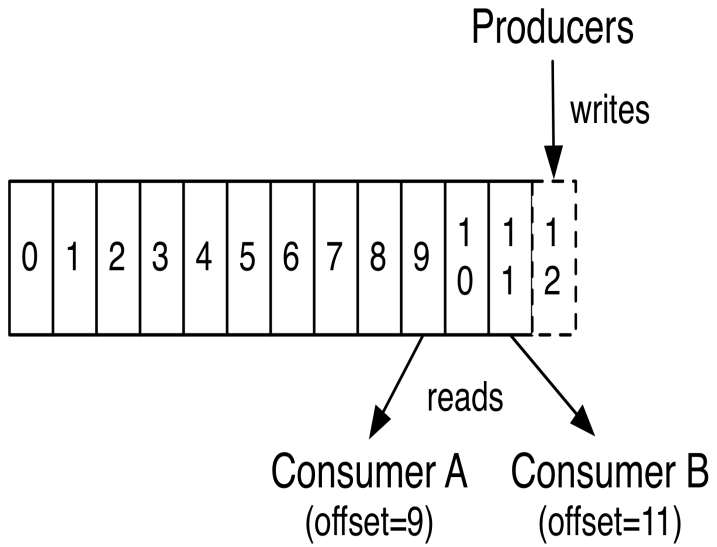
- Each consumer within a consumer group will retrieve data from exclusive partitions
  -> Guarantee each new message will be processed by only one consumer of a consumer group
- Example: NOTIFICATION_GROUP, DATABASE_PERSIST_GROUP
- **Note**: If consumers > partitions, some consumers might not receive anything



Producer and Consumer (5)

Each consumer group has **Consumer Offsets**, containing current reading offsets

- After retrieving data successfully, the consumer will update the corresponding offset
- We can find these offsets at the topic named *__consumer_offset*
- A failure consumer can continue reading current data after restarting



Semantics

There are 3 delivery semantics to commit the new offset:

Exactly one:

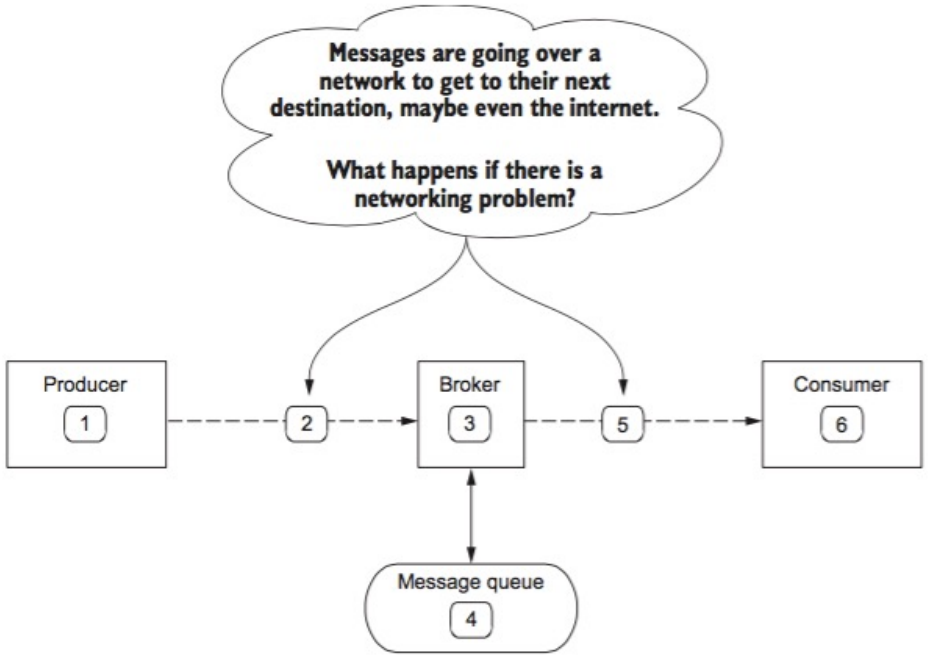
- Guarantees that all messages will always be delivered exactly once
- Only possible through the Streams API in Apache Kafka

At most once:

- Offsets are updated as soon as the message is retrieved
- Will lose current message if there are any exceptions while processing

At least once (*preferred*):

- Offsets are updated after the message is processed successfully
- If there are any errors, the message will be read again until operation successfully -> can cause infinite loop
- A message can be processed twice or more -> Cause multiple effects (ex: duplicated notifications, emails)



Kafka advantages

- High-throughput
- Low latency
- Fault-Tolerant
- Durability

Kafka architecture:

- Scalability
- Distributed
- Message Broker capabilities
- High concurrency
- By default persistent
- Consumer friendly
- Batch handling capable (ETL like functionality)

Kafka operations with commands:

- Variety of use cases
- Real-time handling

Kafka disadvantages

- No complete set of monitoring tools
- Issues with message tweaking
  - It can perform quite well if the message is unchanged because it uses the capabilities of the system
- Not support wildcard topic selection

Kafka monitoring – methods & tools:

- Lack of pace
  - There can be a problem because of the lack of pace, while API's which are needed by other languages are maintained by different individuals and corporates
- Reduces performance
  - The brokers and consumers start compressing these messages as the size increases
- Behaves clumsy
  - when the number of queues in a Kafka cluster increases
- Lacks some messaging paradigms
  - Some of the messaging paradigms are missing in Kafka like request/reply, point-to-point queues

Kafka applications

- **Metrics**: Kafka is often used for operational monitoring data; This involves aggregating statistics from distributed applications to produce centralized feeds of operational data
- **Log Aggregation Solution**: Kafka can be used across an organization to collect logs from multiple services and make them available in a standard format to multiple consumers
- **Stream Processing**: Popular frameworks such as Storm and Spark Streaming read data from a topic, processes it, and write processed data to a new topic where it becomes available for users and applications; Kafka's strong durability is also very useful in the context of stream processing

Kafka summary