APACHE KAFKA

If you don't drive your business,
you will be driven out of business

What is Kafka ?

- An open source apache project initially developed at LinkedIn
- Distributed publish-subscribe messaging system
- Designed for processing of real time activity stream data e.g. logs, metrics collections
- Written in Scala
- Features
  - Persistent messaging
  - High-throughput
  - Supports both queue and topic semantics
  - Uses Zookeeper for forming a cluster of nodes (producer/consumer/broker)

  and many more…
- Multi-language support for Publish/Consumer API (Scala, Java, Ruby, Python, C++, Go, PHP, etc)

Other log aggregation and real-time messaging

Other “log-aggregation only” systems (e.g. Scribe and Flume) are architected for “push” to drive the data.

- high performance and scale however:
  - Expected end points are large (e.g. Hadoop)
  - End points can’t have lots of business logic in real-time because they have to consume as fast as data is pushed to them… accepting the data is their main job

- Messaging Systems (e.g. RabitMQ, ActiveMQ)
  - Does not scale
    - No API for batching, transactional (broker retains consumers stream position)
    - No message persistence means multiple consumers over time are impossible, limiting architecture

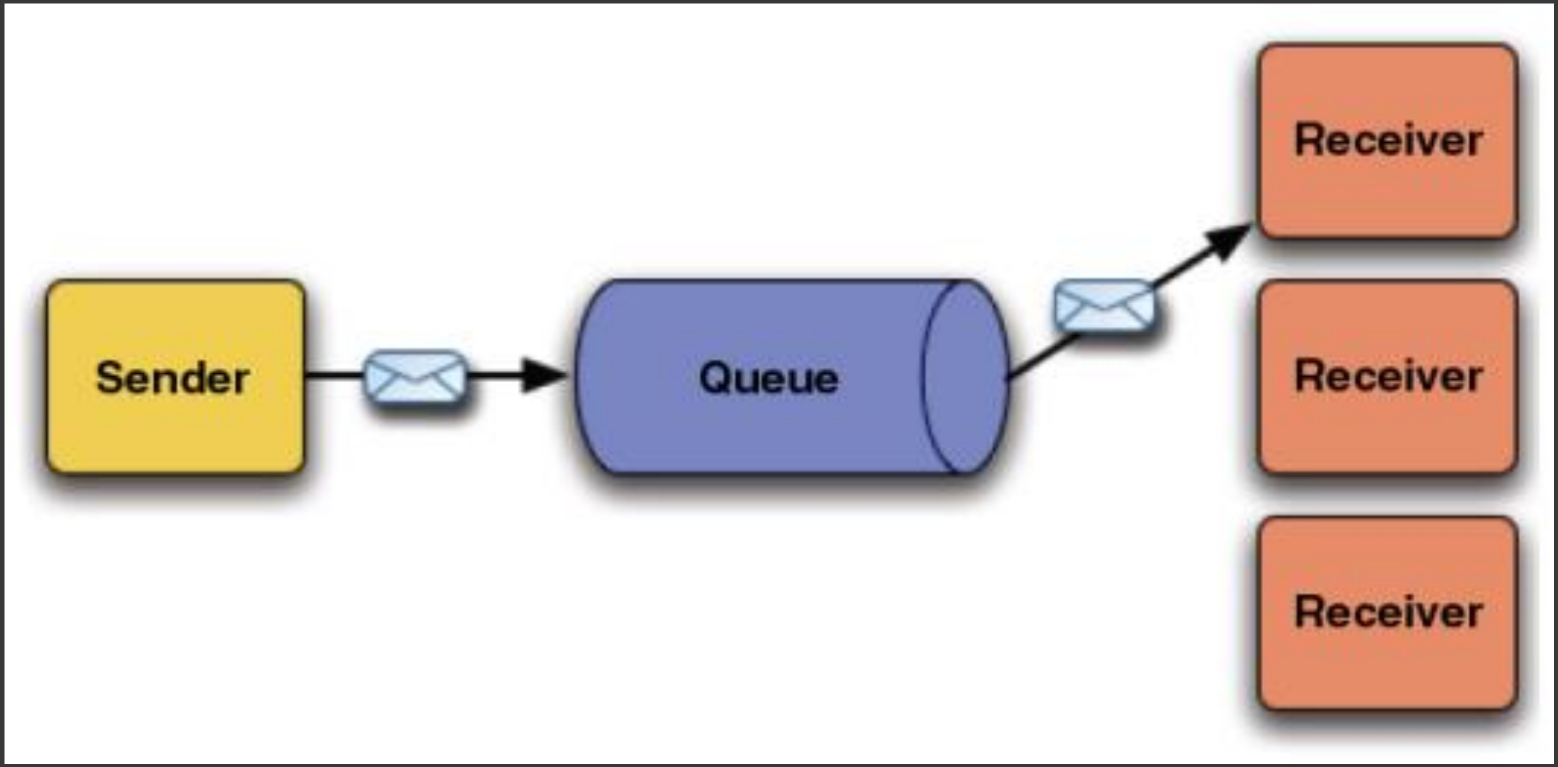
Positioning

Should I use Kafka ?

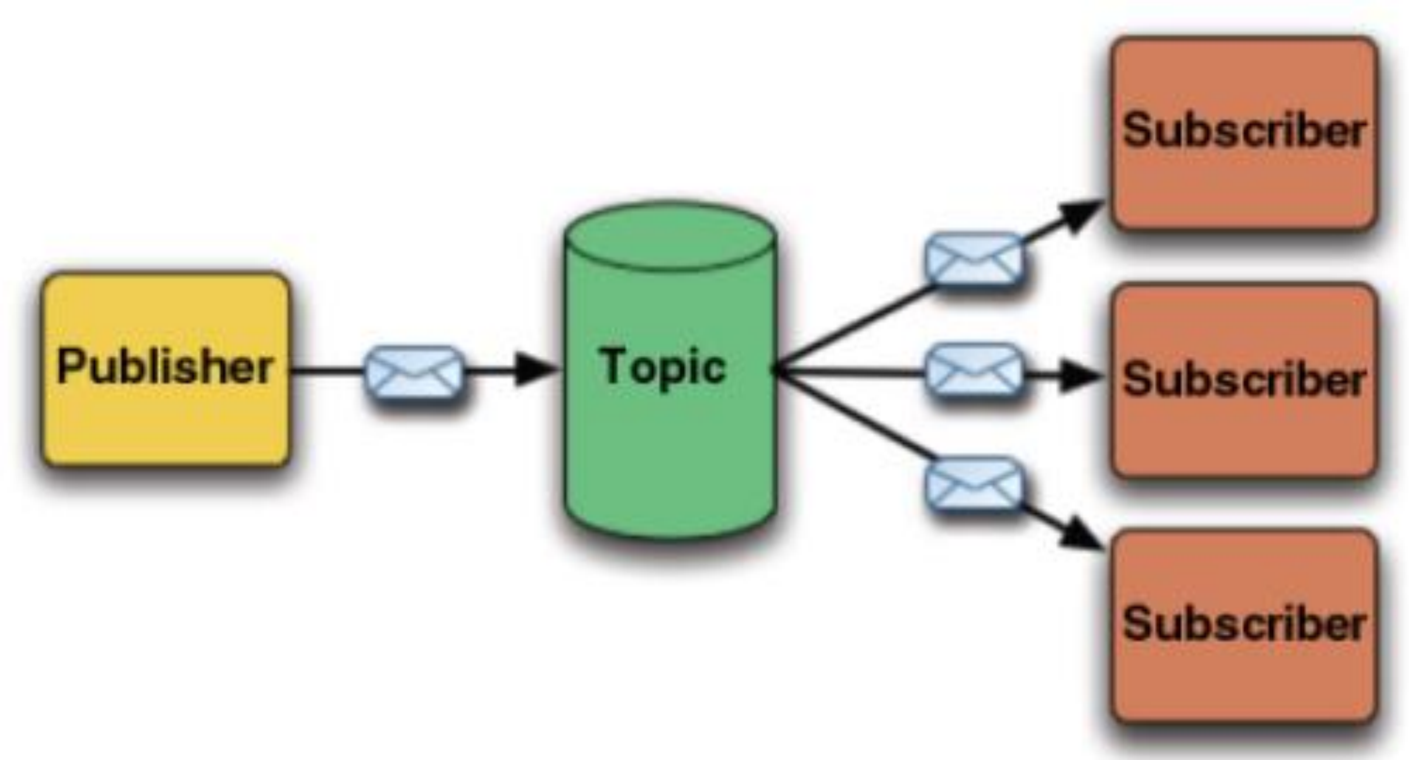
- For really large file transfers?
    - Probably not, it’s designed for “messages” not really for large files. If you need to ship large files, consider good file transfer, or breaking up the files and reading per line to move to Kafka.
- As a replacement for MQ/Rabbit/Tibco
    - Probably. Performance Numbers are drastically superior. Also gives the ability for transient consumers. Handles failures pretty well.
- If security on the broker and across the wire is important?
    - Not right now. We can’t really enforce much in the way of security. (KAFKA-1682)
- To do transformations of data?
    - Not really by itself

Before going into Kafka Specifics Let's recall basic concepts of Messaging System

Point to Point Messaging (Queue)

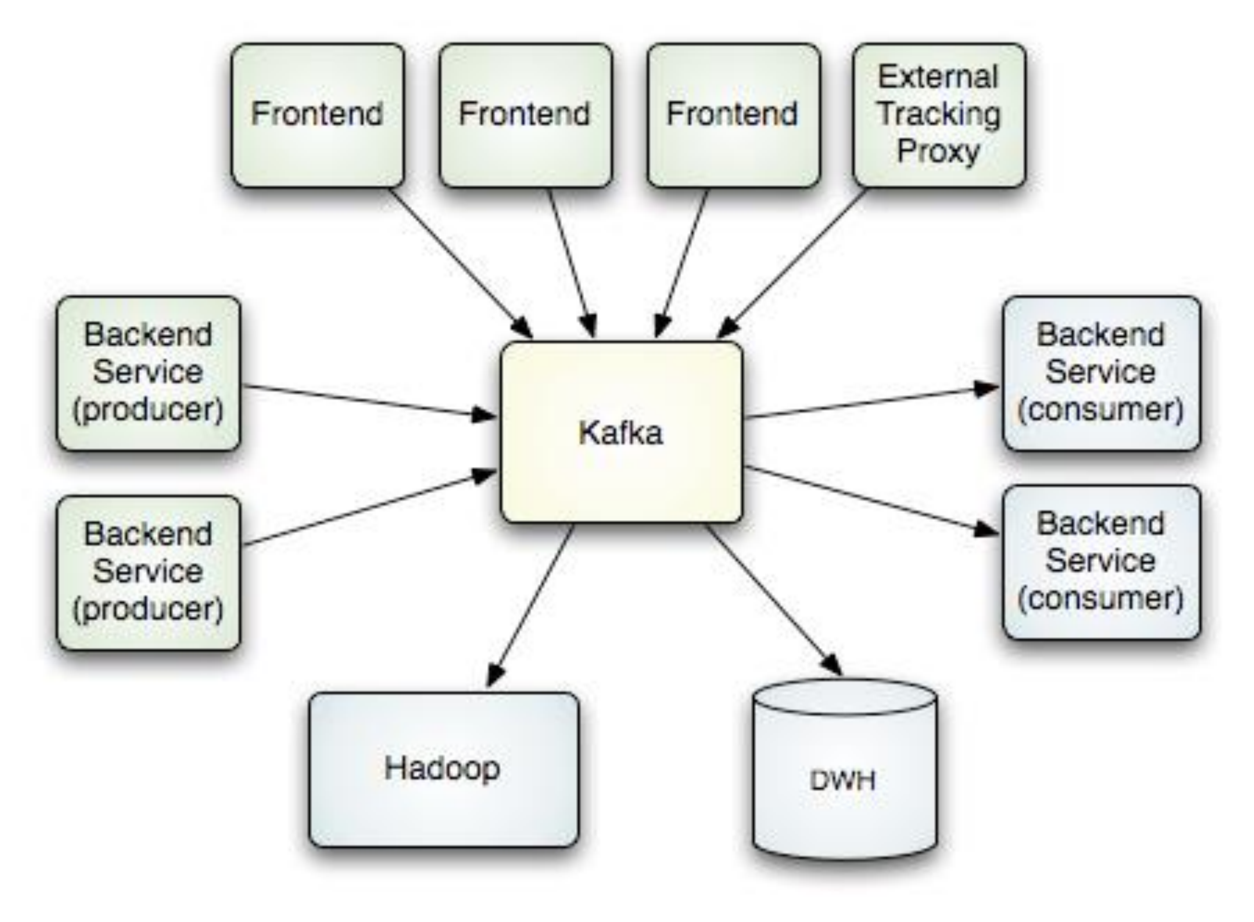


Publish-Subscribe Messaging (Topic)



How Kafka works

Kafka allows sources to push data without worrying about what clients are reading it. Note that producer push, and consumers pull. Kafka itself is a cluster of brokers, which handles both persisting data to disk and serving that data to consumer requests.

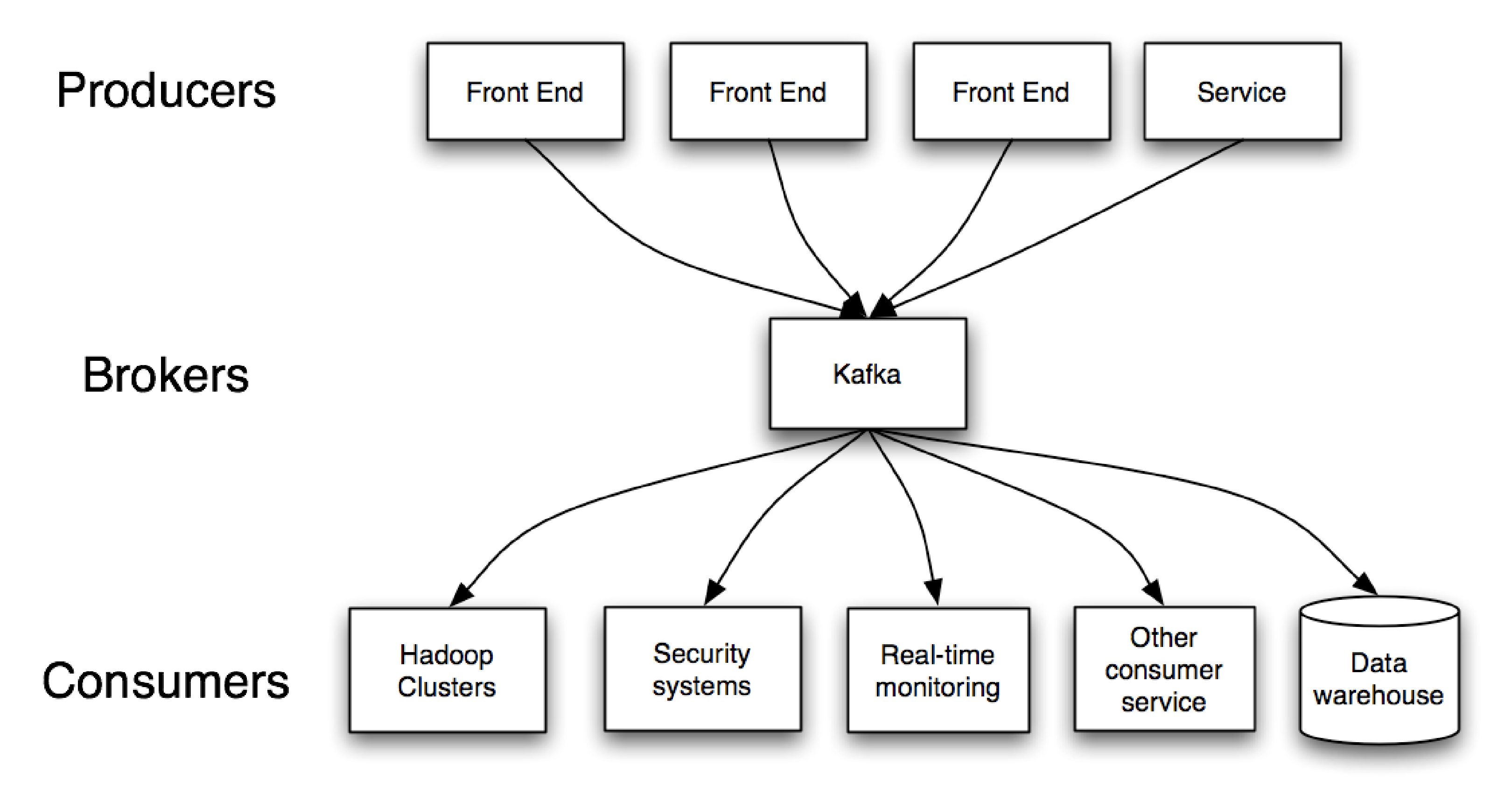

Frontend
Frontend
Frontend
External Tracking Proxy
Backend Service (producer)
Backend Service (producer)
Kafka
Backend Service (consumer)
Backend Service (consumer)
Hadoop
DWH

Key terminology

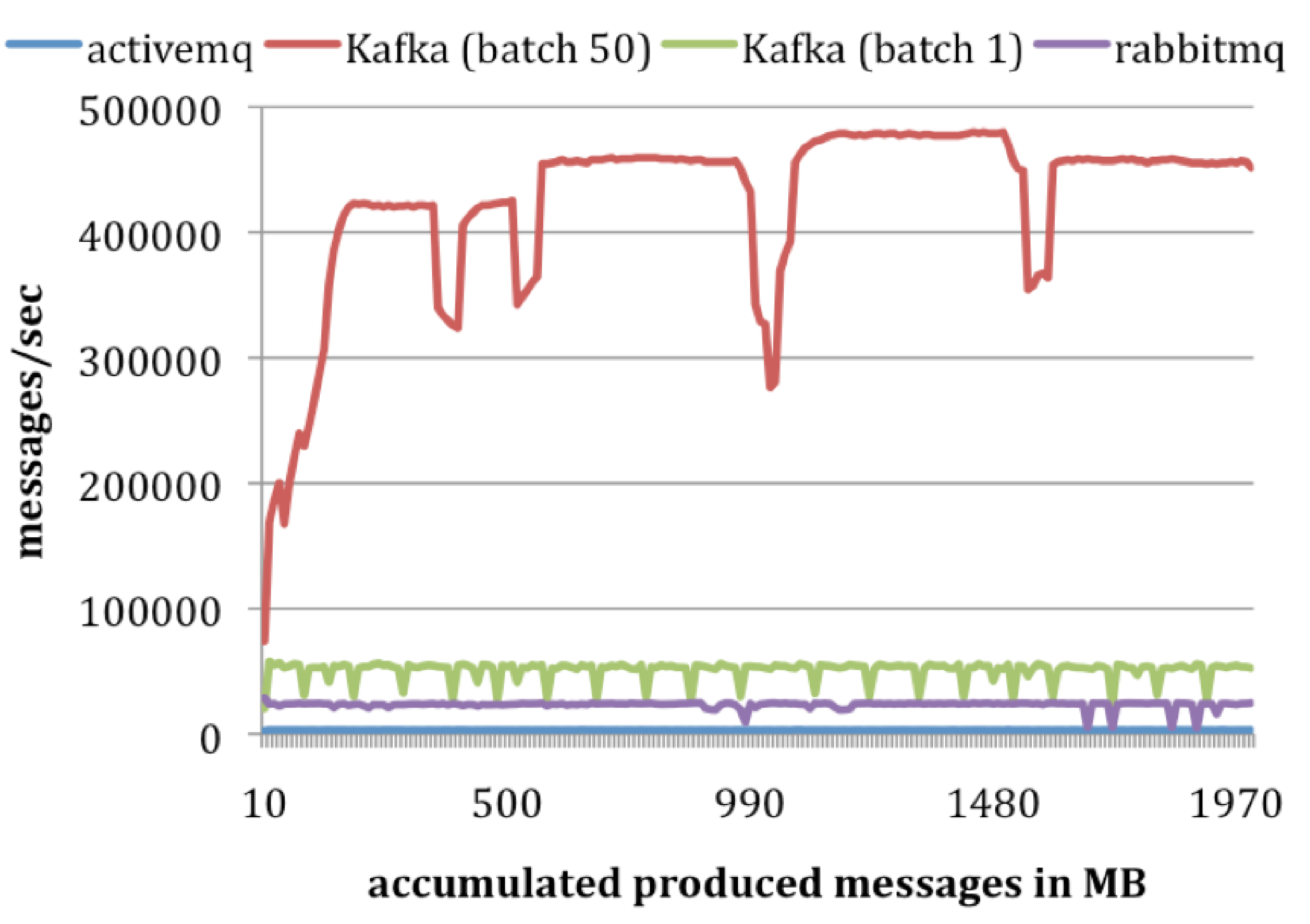
- ❑ Kafka maintains feeds of messages in categories called ***topics***.
- ❑ Processes that publish messages to a Kafka topic are called ***producers***.
- ❑ Processes that subscribe to topics and process the feed of published messages are called ***consumers***.
- ❑ Kafka is run as a cluster comprised of one or more servers each of which is called a ***broker***.

- ❑ Communication between all components is done via a high performance simple binary API over TCP protocol (which you don’t really have to worry too much about)

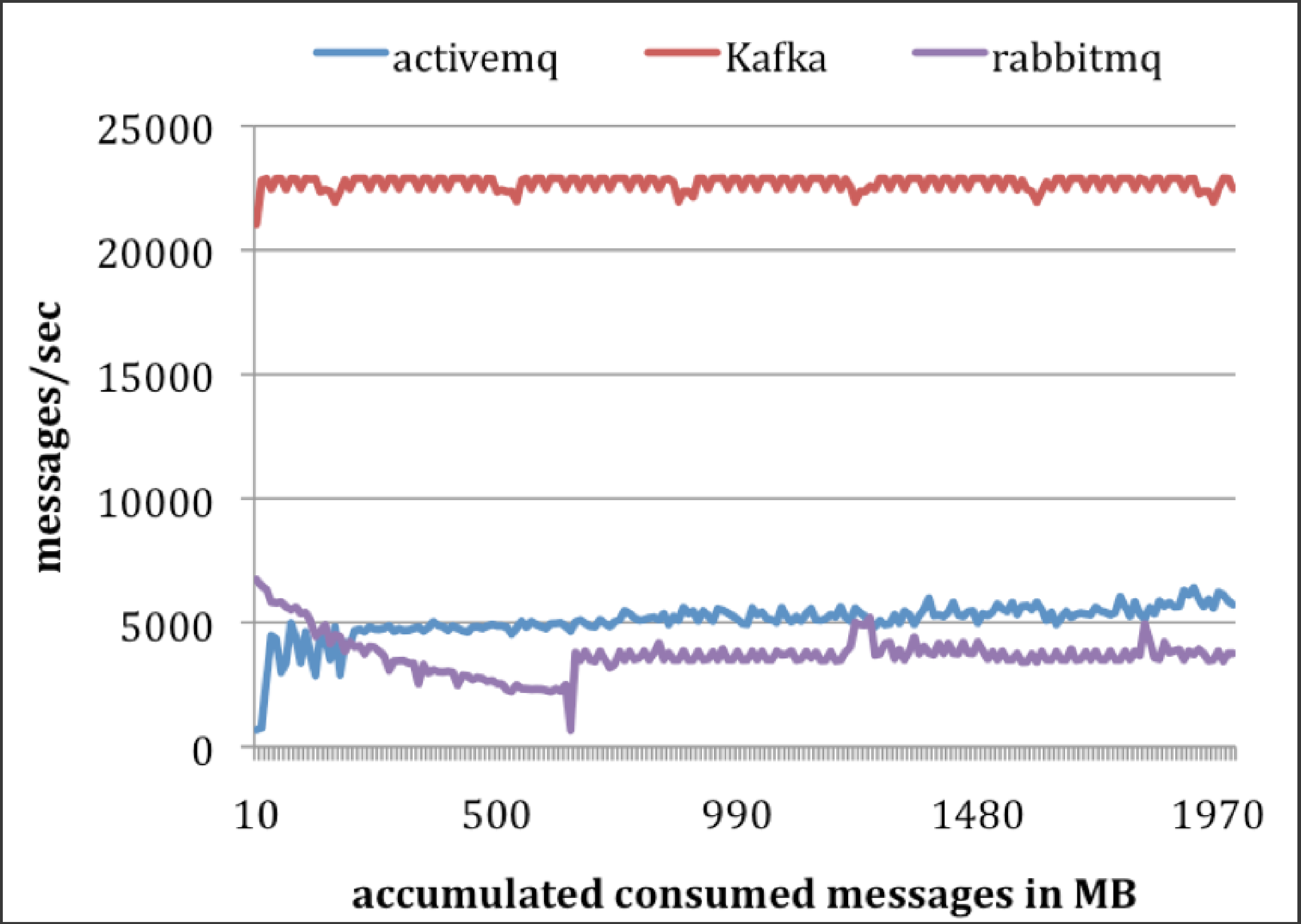
- Kafka is a specialized system and overlaps use cases for both offline and real-time log processing.



Producer Performance & Scale



Consumer Performance & Scale

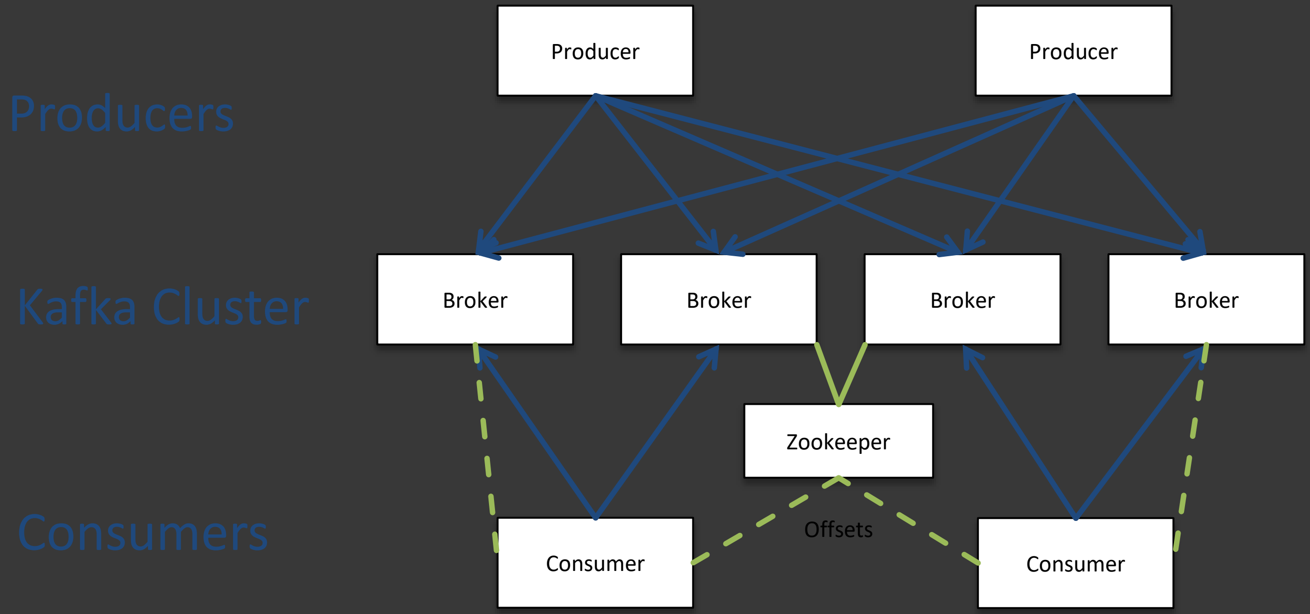


Efficiency

- Kafka achieves it’s high throughput and low latency primarily from two key concepts
- 1) Batching of individual messages to amortize network overhead and append/consume chunks together
- 2) Zero copy I/O using sendfile (Java’s NIO FileChannel transferTo method).
  - Implements linux sendfile() system call which skips unnecessary copies
  - Heavily relies on Linux PageCache
    - The I/O scheduler will batch together consecutive small writes into bigger physical writes which improves throughput.
    - The I/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves throughput.
    - It automatically uses all the free memory on the machine

Architecture

Producers can write to really any broker in the cluster and consumers can do the same.

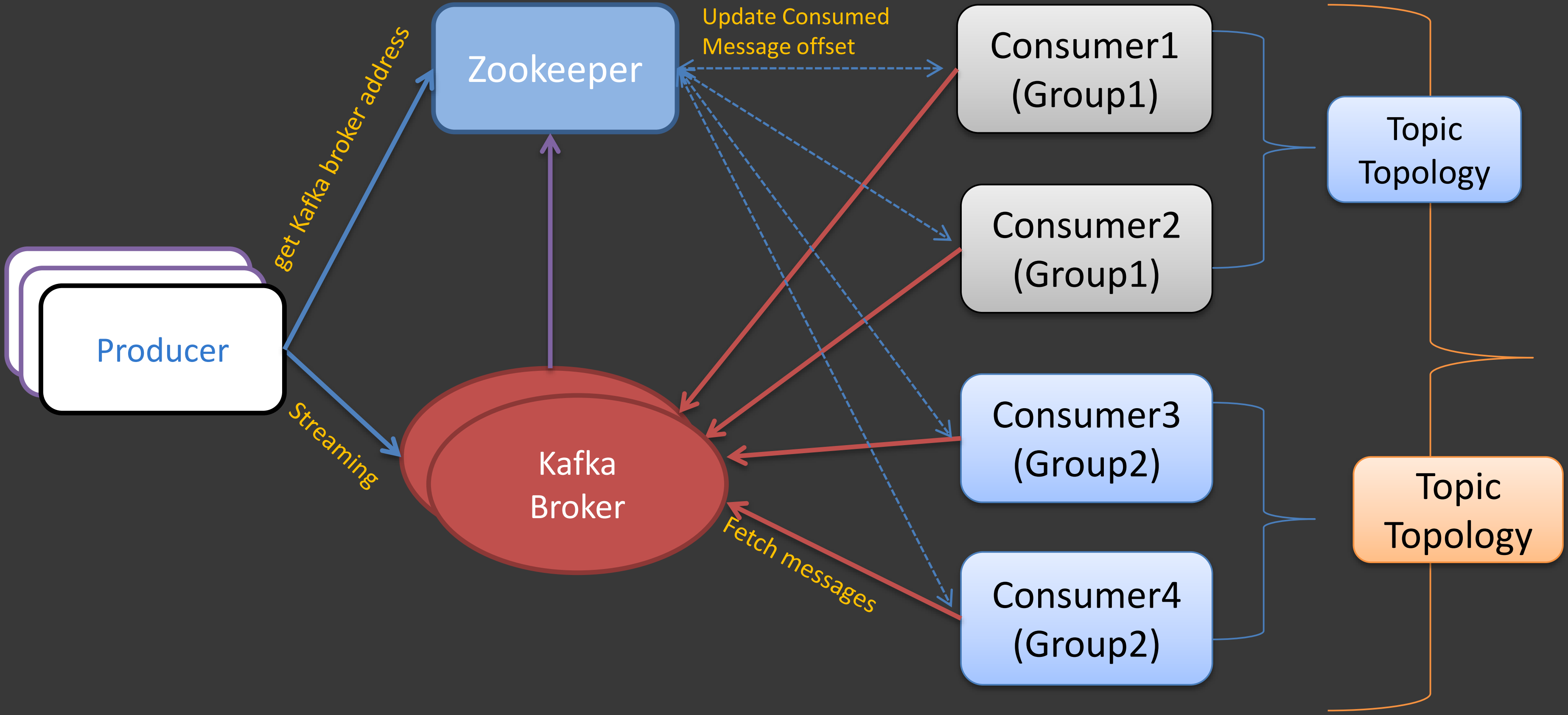


Producers rely on Zookeeper for three basic things:

(1) detecting the addition and the removal of brokers and consumers,
(2) Triggering a rebalance process in each consumer when the above events happen, and
(3) maintaining the consumption relationship and keeping track of the consumed offset of each partition. Specifically, when each broker or consumer starts up, it stores its information in a broker or consumer registry in Zookeeper. The broker registry contains the broker's host name and port, and the set of topics and partitions stored on it

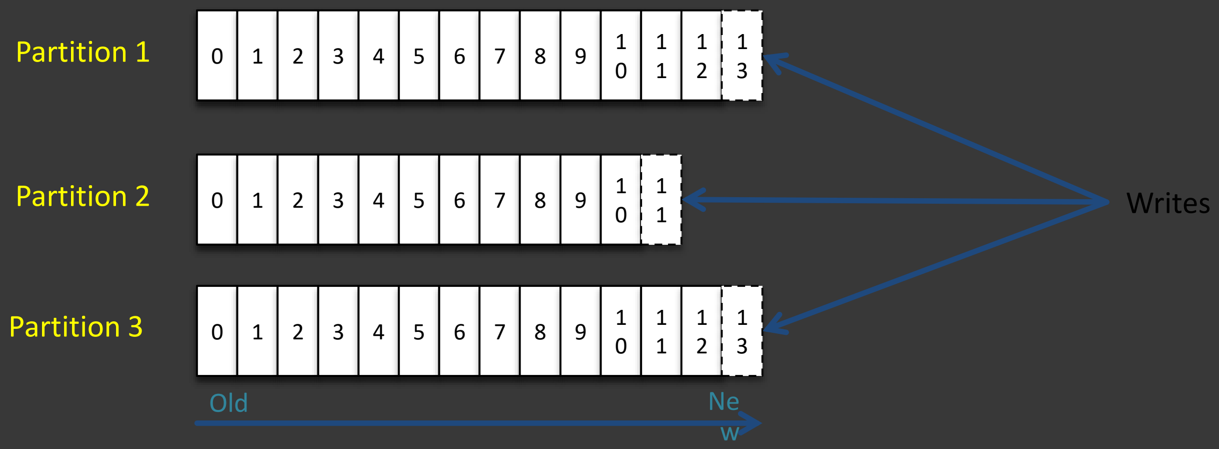
Real time transfer

Broker does not ***Push*** messages to Consumer, Consumer ***Polls*** messages from Broker.



Topics - Partitions

- Topics are broken up into ordered commit logs called partitions.
- Each message in a partition is assigned a sequential id called an offset.
- Data is retained for a configurable period of time*



Partition 1: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

Partition 2: 0 1 2 3 4 5 6 7 8 9 10 11

Partition 3: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

Old

New

Writes

Message Ordering

- Ordering is only guaranteed within a partition for a topic
- To ensure ordering:
  - Group messages in a partition by key (producer)
  - Configure exactly one consumer instance per partition within a consumer group

Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- A consumer instance sees messages in the order they are stored in the log
- For a topic with replication factor N, Kafka can tolerate up to N-1 server failures without “losing” any messages committed to the log

Topics - Replication

- Topics can (and should) be replicated.
- The unit of replication is the partition
- Each partition in a topic has 1 leader and 0 or more replicas.
- A replica is deemed to be “in-sync” if
  - The replica can communicate with Zookeeper
  - The replica is not “too far” behind the leader (configurable)
- The group of in-sync replicas for a partition is called the ***ISR*** (In-Sync Replicas)
- The Replication factor cannot be lowered

Topics - Replication

- Durability can be configured with the producer configuration ***request.required.acks***
  - ***0*** The producer never waits for an ack
  - ***1*** The producer gets an ack after the leader replica has received the data
  - ***-1*** The producer gets an ack after all ISRs receive the data
- Minimum available ISR can also be configured such that an error is returned if enough replicas are not available to replicate data

Durable Writes

- Producers can choose to *trade* throughput for durability of writes:

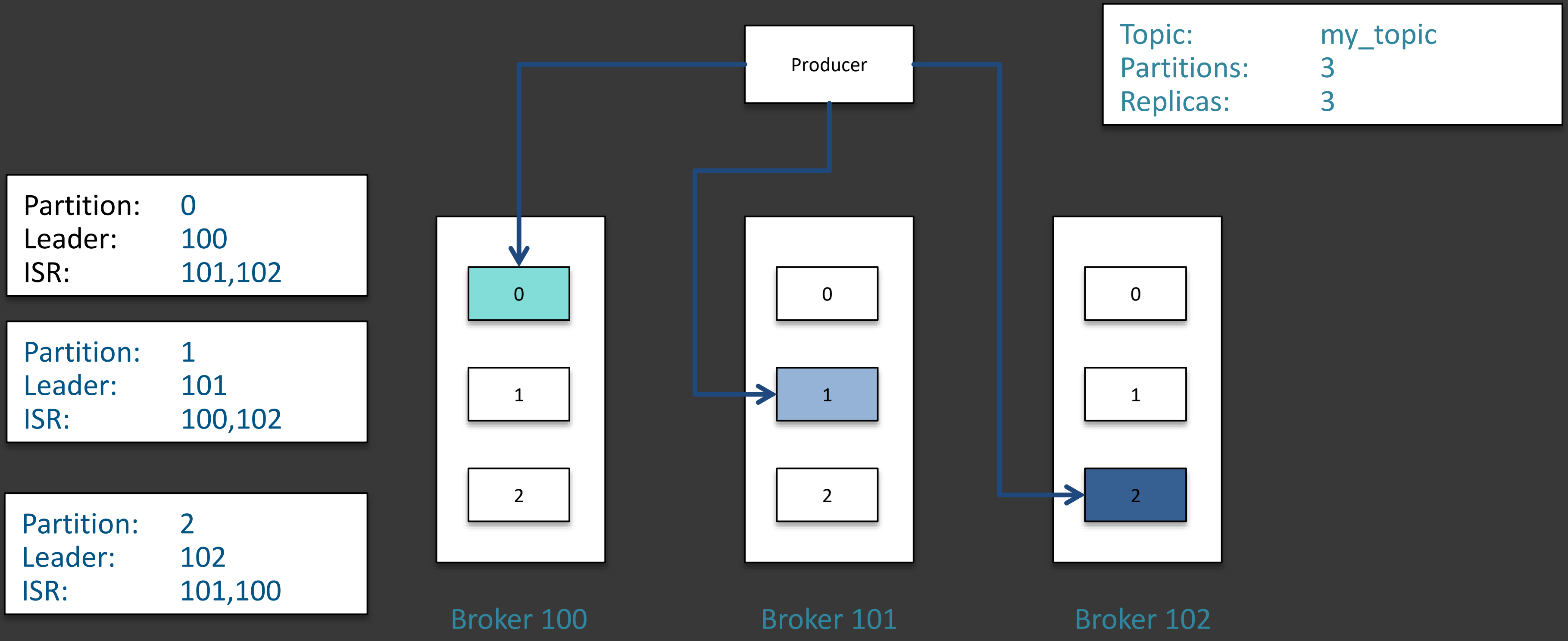
Durability	Behaviour	Per Event Latency	Required Acknowledgements (request.required.acks)
Highest	ACK all ISRs have received	Highest	-1
Medium	ACK once the leader has received	Medium	1
Lowest	No ACKs required	Lowest	0

- A sane configuration:

Property	Value
replication	3
min.insync.replicas	2
request.required.acks	-1

Producer – Load Balancing and ISRs

Load can be distributed typically by “round-robin”



Topic:	my_topic
Partitions:	3
Replicas:	3

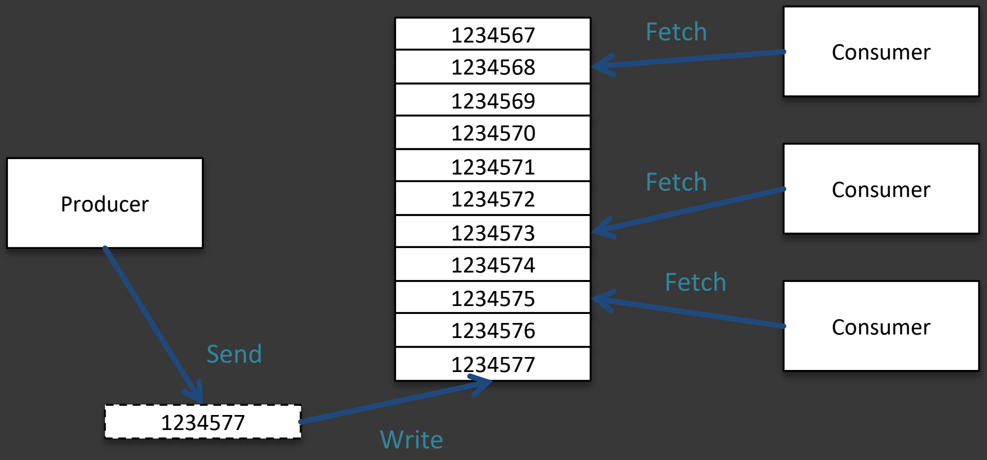
Partition:	0
Leader:	100
ISR:	101,102

Partition:	1
Leader:	101
ISR:	100,102

Partition:	2
Leader:	102
ISR:	101,100

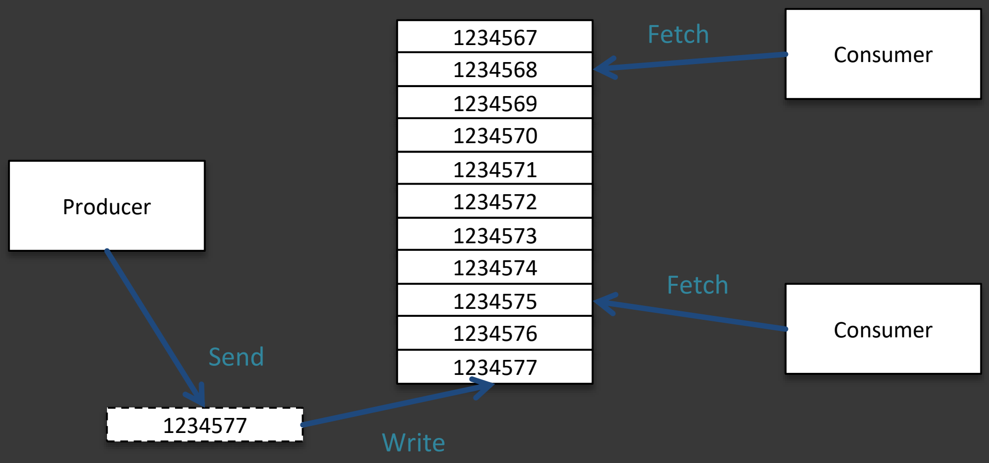
Consumer

- Multiple Consumers can read from the same topic
- Each Consumer is responsible for managing it's own offset
- Messages stay on Kafka…they are not removed after they are consumed



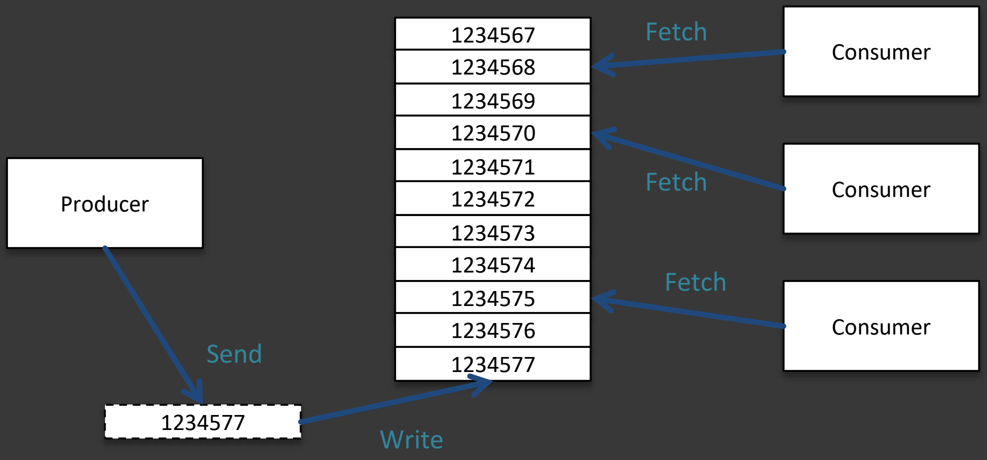
Consumer

- Consumers can go away



Consumer

- And then come back



Kafka Installation

- Download
  - http://kafka.apache.org/downloads.html

- Untar it

```
> tar -xzf kafka_<version>.tgz
> cd kafka_<version>
```

Start Servers

- Start the Zookeeper server

```
> bin/zookeeper-server-start.sh config/zookeeper.properties
```

Pre-requisite: Zookeeper should be up and running.

- Now Start the Kafka Server

```
> bin/kafka-server-start.sh config/server.properties
```

Create/List Topics

- Create a topic

```
> bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 1 --partitions 1 --topic
test
```

- List down all topics

```
> bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Output: test

Producer

- Send some Messages

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

```
Now type on console:
This is a message
This is another message
```

Consumer

- Receive some Messages

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic test --from-beginning
This is a message
This is another message
```

Multi-Broker Cluster

- Copy configs

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

- Changes in the config files.

config/server-1.properties:

```
broker.id=1
port=9093
log.dir=/tmp/kafka-logs-1
```

config/server-2.properties:

```
broker.id=2
port=9094
log.dir=/tmp/kafka-logs-2
```

Start with New Nodes

- Start other Nodes with new configs

```
> bin/kafka-server-start.sh config/server-1.properties &
> bin/kafka-server-start.sh config/server-2.properties &
```

- Create a new topic with replication factor as 3

```
> bin/kafka-topics.sh --create --zookeeper
localhost:2181 --replication-factor 3 --partitions
1 --topic my-replicated-topic
```

- List down the all topics

```
> bin/kafka-topics.sh --describe --zookeeper
localhost:2181 --topic my-replicated-topic
Topic:my-replicated-topic PartitionCount:1 ReplicationFactor:3 Configs:
Topic: my-replicated-topic Partition: 0 Leader: 1 Replicas: 1,2,0 Isr: 1,2,0
```

Multi broker messages

- Start other Nodes with new configs

```
bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-replicated-topic
 ...
 my test message 1
 my test message 2
 ^C
```

- Now consume this message:

```
bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic
 ...
 my test message 1
 my test message 2
```

- Now let's test out fault-tolerance. Kill the broker acting as leader for this topic's only partition:

```
> pkill -9 -f server-1.properties
```

Leadership should switch to one of the slaves:

```
> bin/kafka-list-topic.sh --zookeeper localhost:2181
topic: my-replicated-topic    partition: 0        leader: 2   replicas: 1,2,0        isr: 2
```

Multi broker messages

- And the messages should still be available for consumption even though the leader that took the writes originally is down:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --from-beginning --topic my-replicated-topic
...
my test message 1
my test message 2
^C
```

Producer API

```
/**
 *  V: type of the message
 *  K: type of the optional key associated with the message
 */
class kafka.javaapi.producer.Producer<K,V>
{
  public Producer(ProducerConfig config);

  /**
   * Sends the data to a single topic, partitioned by key,
 * @param message the producer data object that encapsulates the topic, key and message data
   */
  public void send(KeyedMessage<K,V> message);

  /**
   * Use this API to send data to multiple topics
   * @param messages list of producer data objects that encapsulate the topic, key and message data
   */
  public void send(List<KeyedMessage<K,V>> messages);
  /**
   * Close API to close the producer pool connections to all Kafka brokers.
   */
  public void close();
}
```

Consumer API

```
class kafka.javaapi.consumer.SimpleConsumer {
 /**
  *  Fetch a set of messages from a topic.
  *  @param request specifies the topic name, topic partition, starting byte offset, maximum bytes to be fetched.
  *  @return a set of fetched messages
  */
 public FetchResponse fetch(request: kafka.javaapi.FetchRequest);
 /**
  *  Fetch metadata for a sequence of topics.
  *  @param request specifies the versionId, clientId, sequence of topics.
  *  @return metadata for each topic in the request.
  */
 public kafka.javaapi.TopicMetadataResponse send(request: kafka.javaapi.TopicMetadataRequest);
 /**
  *  Get a list of valid offsets (up to maxSize) before the given time.
  *  @param request a [[kafka.javaapi.OffsetRequest]] object.
  *  @return a [[kafka.javaapi.OffsetResponse]] object.
  */
 public kafka.javaapi.OffsetResponse getOffsetsBefore(request: OffsetRequest);
 /**
  * Close the SimpleConsumer.
  */
 public void close();
}
```

THANK YOU