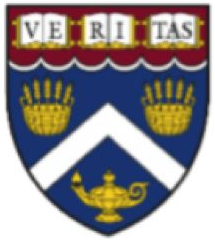
CSCI E-88 Principles Of Big Data Processing

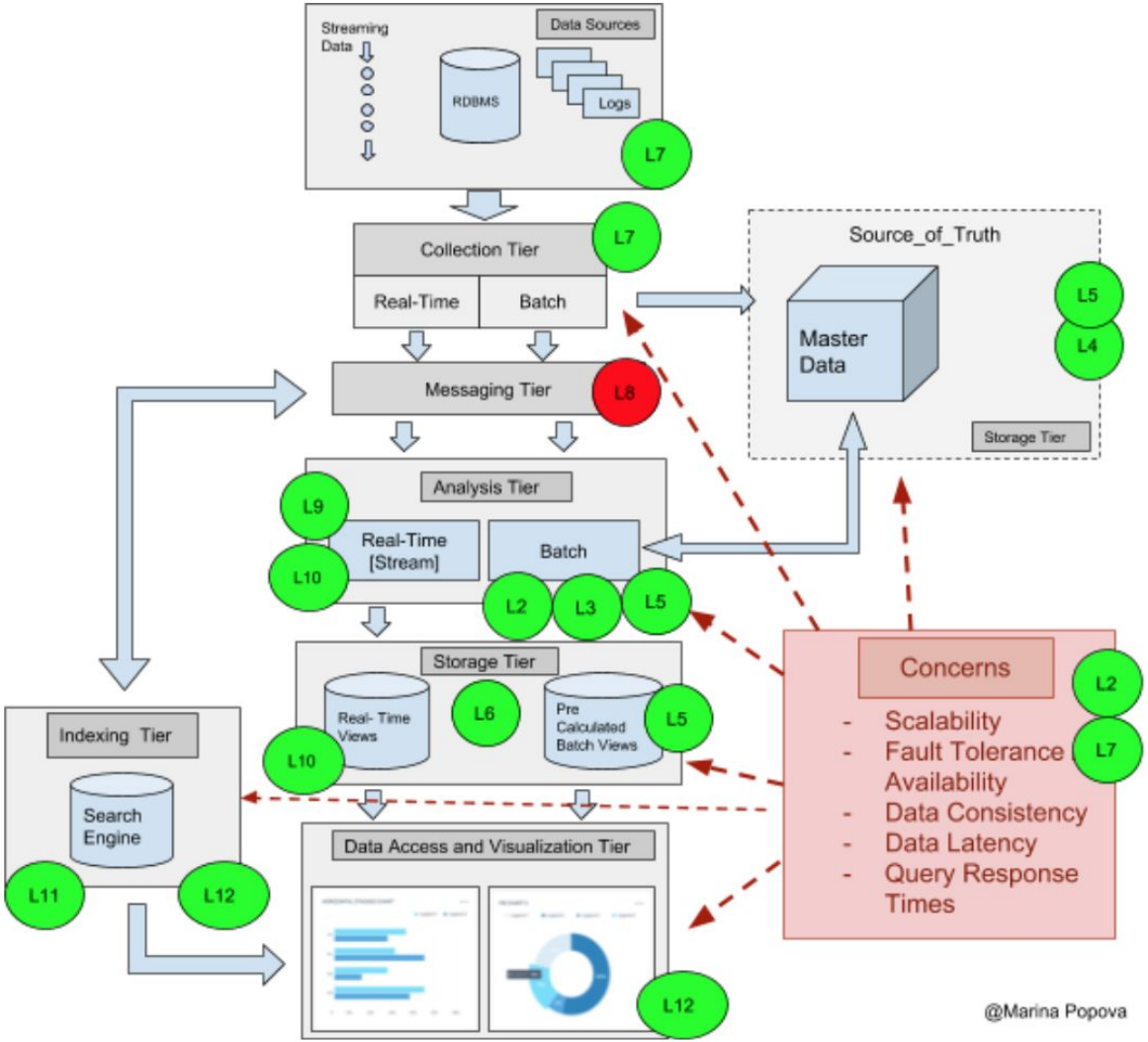
Harvard University Extension, Fall 2019

Marina Popova



Lecture 8 - Messaging Tier and Kafka

Where Are We?


Streaming Data
Data Sources
RDBMS
Logs
L7
Collection Tier
L7
Real-Time
Batch
Source_of_Truth
Master Data
L5
L4
Storage Tier
Messaging Tier
L8
Analysis Tier
L9
Real-Time [Stream]
Batch
L10
L2
L3
L5
Storage Tier
Real- Time Views
L6
Pre Calculated Batch Views
L5
L10
Indexing Tier
Search Engine
L11
L12
Data Access and Visualization Tier
L12
Concerns
- Scalability
- Fault Tolerance
- Availability
- Data Consistency
- Data Latency
- Query Response Times
L2
L7
@Marina Popova

Administrative details

- Midterm Quiz info
  - Open book
  - 1 hr time limit
  - Multiple-choice answers

Agenda

- Messaging Systems - general concepts
- Kafka - introduction and deeper dive
- Kafka - Producer and Consumer APIs

Message Systems

Concepts and Alphabet Soup - Demystified

Alphabet and acronyms soup:
MOM vs AMQP vs Message Brokers vs JMS

Good references: https://en.wikipedia.org/wiki/Message_queue
https://stackoverflow.com/questions/13202200/message-broker-vs-mom-message-oriented-middleware/36999850

Ref: http://iopscience.iop.org/article/10.1088/1742-6596/608/1/012038/meta by L Magnoni

Message Systems

MOM: Message-Oriented-Middleware

MOM is a loosely coupled communication solution which minimizes producer and consumer dependencies. The biggest advantage of MOM architecture/decision is **decoupling of the components**.

It is an approach to design a distributed system where there are many components that have to share info (messages) among them.

A **message** is a discrete piece of information, shared between such components

Like many other technologies, messaging is based on some **basic concepts and properties** which are shared among all the different specific implementations - which we will review next.

MOM products used to be quite large and complex: CORBA, JMS, TIBCO, WebsphereMQ, etc. and tried to do a lot more than simply deliver messages

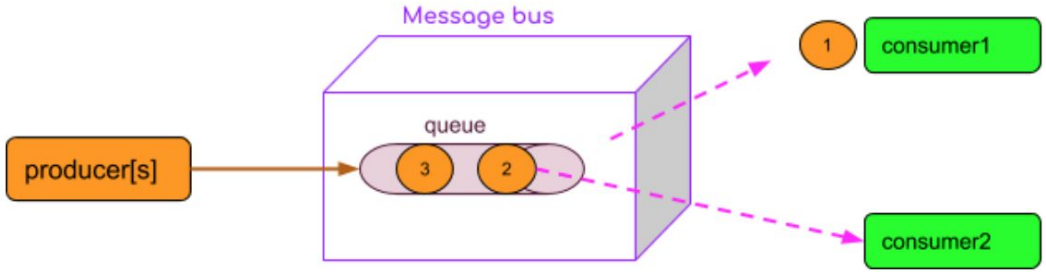
Message Systems - Properties:

- **Communication model: point-to-point or pub-sub**
- **Delivery policies**: do we need to guarantee that a message is delivered at least once, at most once, exactly once?
- **Durability**/**Persistence**, which is the ability to save message on persistent storage, such as file-system or database
- **Fail-over**, which allow clients to automatically reconnect in case of broker failure
- **Security policies** - which applications should have access to these messages?
- **Message purging policies** - queues or messages may have a "time to live"
- **Message filtering** - some systems support filtering data by pre-specified criteria
- **Ordering**, to deliver messages in the order they are produced
- **Transaction**, the ability to consider multiple requests as part of a distributed transaction, with roll-back options
- **Clustering/ Topology/ Routing**: which is the possibility to create network of message brokers for high-availability and load-balancing
- **Batching policies** - should messages be delivered immediately? Or should the system wait a bit and try to deliver many messages at once?
- **Receipt notification** - A publisher may need to know when some or all subscribers have received a message.

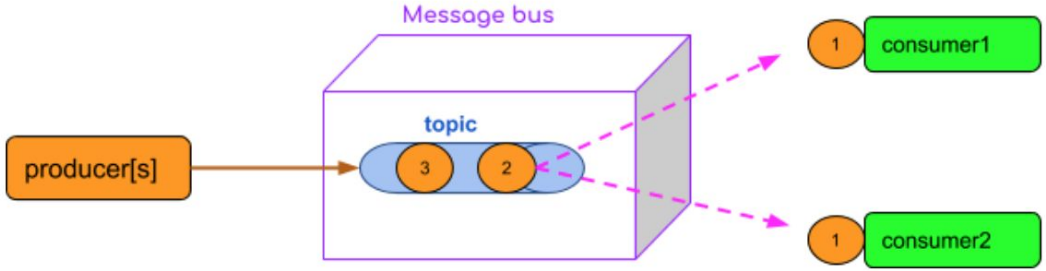
Different messaging systems may provide different interpretation for the same features. Many other unique broker specific features exist, but their usage imply hard coupling the application with a specific broker flavour.

Messaging systems: communication models:

Point-to-point:



Pub-Sub:



Message Systems - Basic Concepts

Communication models:
point-to-point (queue) and publish-and-subscribe (topics)

Messaging systems support different communication models, each one defining how the information is exchanged among producer and consumer. The most common communication models are point-to-point and publish-and-subscribe.

Point-to-point/ message queues

- implemented using queues; **Message queues provide an asynchronous communications protocol**, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time.
- if no consumer exists when the information is produced, the message is kept in the channel for later delivery
- if there are multiple consumers the message is delivered to one consumer only
- Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue

Message Systems

Publish/subscribe

- Messages are not sent directly to specific receivers, but rather grouped into named channels, called "topics"
- Consumers interested in receiving messages of this type express their interest by "subscribing" to the topic
- if no consumer exists the message is discarded (or persisted if durable subscribers are used)
- in case of multiple consumers the message system delivers it to each of them
- There are models for **durable and non-durable subscribers**
    - Durables subscriber does not have to be active/alive when messages are sent to the topic - the MOM system will deliver all missed messages to such subscribers when they become alive
    - Non-durable subscribers can only receive messages from the topic when they are alive

There can be other, more complex delivery options at protocol level (e.g. exchange/nodes from AMQP) and many others are middleware-specific.

Message Systems: APIs and Protocols

To implement a message-oriented-middleware in an inter-operable way we need a set of specific rules for how the messages are published, consumed, how the acknowledgement will work, the lifetime of a message until it is consumed, the persistence of a message, etc.

this is what **wire protocols** are for

AMQP: Advanced Message Queueing Protocol - is one of such protocols.

From WIkipedia:

AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns.

http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html

A **binary protocol** is a protocol which is intended to be read by a machine rather than a human being, as opposed to a plain text protocol such as IRC, SMTP, or HTTP/1.1. Binary protocols have the advantage of terseness, which translates into speed of transmission and interpretation.

a Glimpse at a Binary protocol ...

PART 1. TYPES 1.2 Type Encodings

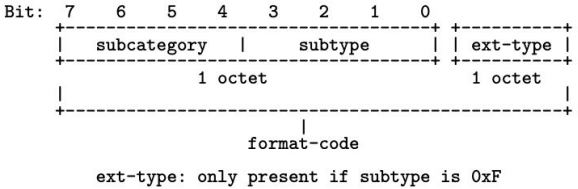
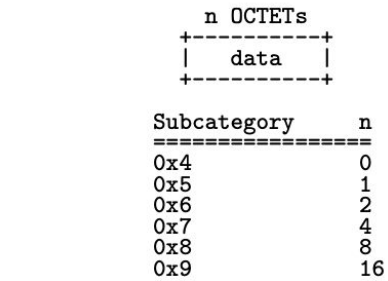


Figure 1.4: Format Code Layout

The following table describes the subcategories of format-codes:

Subcategory	Category	Format
0x4	Fixed Width	Zero octets of data.
0x5	Fixed Width	One octet of data.
0x6	Fixed Width	Two octets of data.
0x7	Fixed Width	Four octets of data.
0x8	Fixed Width	Eight octets of data.
0x9	Fixed Width	Sixteen octets of data.
0xA	Variable Width	One octet of size, 0-255 octets of data.
0xB	Variable Width	Four octets of size, 0-4294967295 octets of data.
0xC	Compound	One octet each of size and count, 0-255 distinctly typed values.
0xD	Compound	Four octets each of size and count, 0-4294967295 distinctly typed values.
0xE	Array	One octet each of size and count, 0-255 uniformly typed values.
0xF	Array	Four octets each of size and count, 0-4294967295 uniformly typed values.



Subcategory	n
0x4	0
0x5	1
0x6	2
0x7	4
0x8	8
0x9	16

Figure 1.6: Layout of Fixed Width Data Encodings

Ref:
http://docs.oasis-open.org/amqp/core/v1.0/csprd01/amqp-core-complete-v1.0-csprd01.pdf

@Marina Popova

Message Systems: APIs and Protocols

AMQP is a wire specification for **asynchronous messaging**

It does not define a wire-level distinction between "clients" and "brokers", the protocol is symmetric.

every **byte** of transmitted data is specified

What does it define?

- type system and type encodings
- peer-to-peer transport protocol which operates over TCP
- messaging layer and requirements for transactional messaging
- Security Layers to provide an authenticated and/or encrypted transport

.

This means that specific libraries can be written in many languages, to run on multiple operating systems and CPU architectures - and this is what makes this protocol a truly interoperable, cross-platform messaging standard.

Message Systems

Is this the only protocol? No

From Wikipidia:

These are the known open protocol specifications that cover the same or similar space as AMQP:

- **Streaming Text Oriented Messaging Protocol (STOMP),** a text-based protocol developed at Codehaus; uses the JMS-like semantics of 'destination'.
- **Extensible Messaging and Presence Protocol (XMPP)** - "At its core, XMPP is a technology for streaming XML over a network"
- **MQTT (Message Queue Telemetry Transport),** a lightweight machine-to-machine publish-subscribe protocol, designed originally by IBM. It is meant for low bandwidth, high-latency networks
- **OpenWire** - used by ActiveMQ

there are others too (RSS, Atom, REST, ...)

Message Systems: JMS vs AMQP

From Wikipedia: **Java Message Service (JMS)**, is often compared to AMQP, as it is the most common messaging system in the Java community. Interesting Refs for JMS: https://www.journaldev.com/9743/jms-messaging-models

JMS	AMQP
JMS is an API specification (part of the Java EE specification) that defines how message producers and consumers are implemented.	**does not have a standard API** (similar to HTTP)
does not guarantee interoperability between implementations, and **the same** JMS-compliant messaging system may need to be deployed on both client and server	In theory, **provides interoperability** as **different AMQP-compliant software** can be deployed on the client and server sides
message format is not specified - JMS has no requirement for how messages are formed and transmitted	**AMQP is a wire-level protocol** specification - message format is specified for each byte

Message Systems: JMS vs AMQP Summary

- AMQP is a wire-level messaging protocol that does not implement the JMS API
- JMS is only an API spec. It doesn't use or mandate any wire protocol
- There are implementations of AMQP protocol that may or may not also implement JMS API
  - For example: RabbitMQ, which is written in Erlang , and is implementing the AMQP, but not JMS
- There are implementations of JMS API that could be using any wire protocol (AMQP or other)
  - For example: Apache ActiveMQ can use any of the following protocols: AMQP, MQTT, OpenWire, REST(HTTP), RSS and Atom, Stomp, WSIF, WS Notification, XMPP.

Message Systems

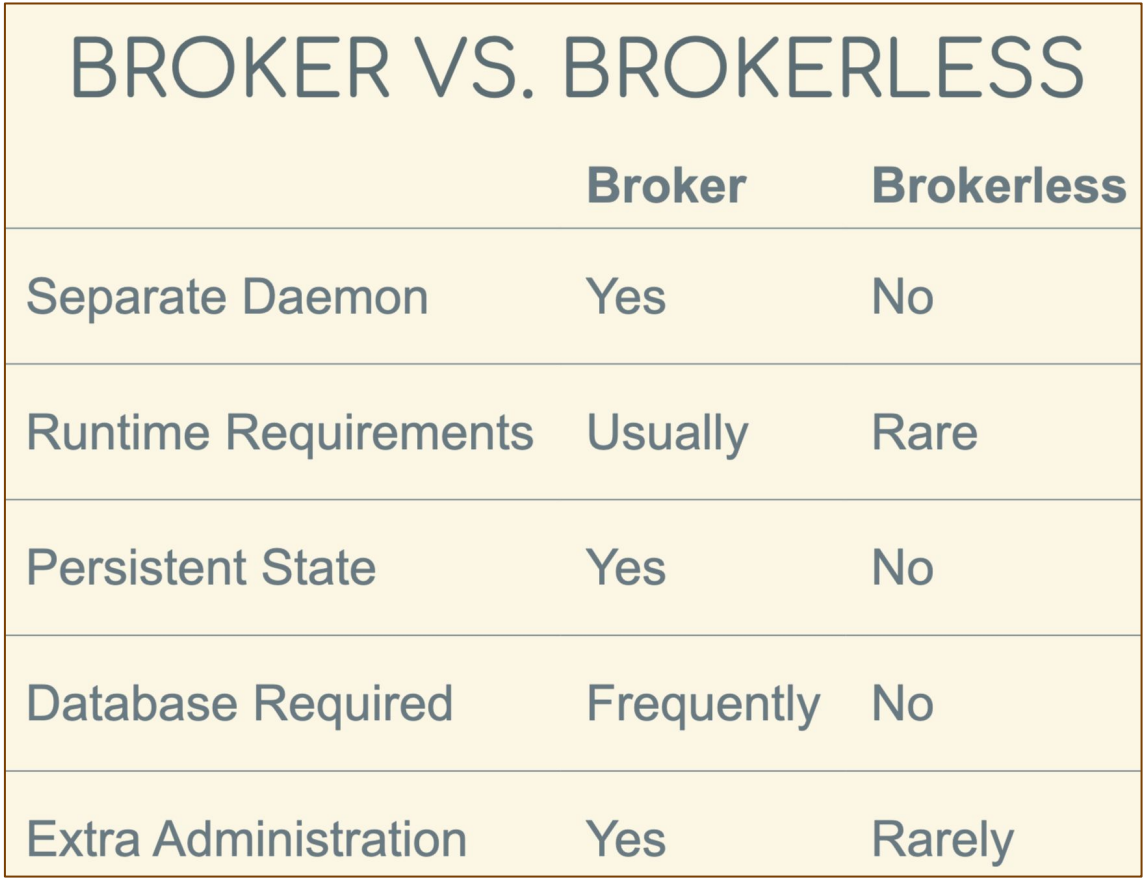
What about message brokers?

There are different types of MOM implementations in terms of **broker usage**: **Broker-based and Brokerless MOMs**

- **Message brokers**, also called **broker-based MOMs**, are the most common implementation of messaging system.
- A message broker is a standalone entity which offers messaging functionality via standard or custom protocols.
- There are many message broker implementations, with different capabilities, protocols, implementation languages, platform support.
- With **broker-based MOMs, all messages go to one central place: broker, and get distributed from there**.
- Message brokers are the most feature-rich type of messaging system, in term of capabilities and protocol support. Brokers can be polyglot, allowing producer and consumer to use different protocol (e.g. sender over AMQP, receiver over STOMP) and they can support message transformation (e.g. transforming message payload from XML to JSON).
- **AMQP is a protocol mainly designed for broker based MOMs** and there are several different Message brokers implementing that protocol, for example RabbitMQ and ActiveMQ.

Broker vs. Broker-less

Ref:
https://staysail.github.io/nng_presentation/nng_presentation.html



@Marina Popova

Message Systems

Brokerless MOMs

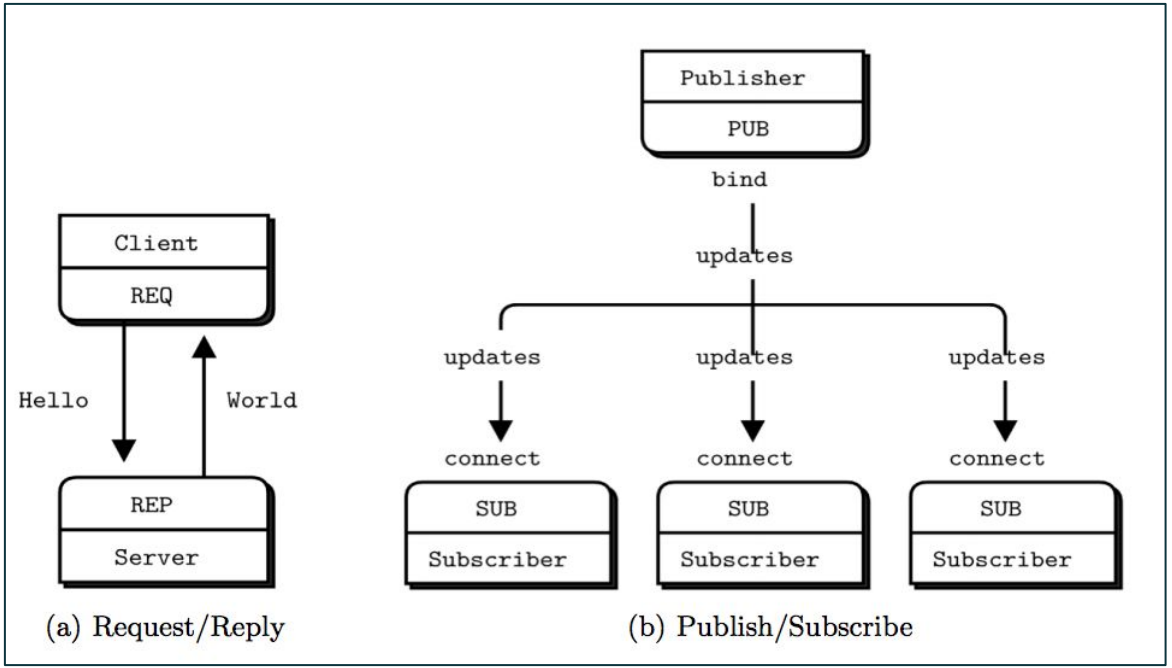
brokerless MOM usually allows for peer to peer messaging (but does not exclude option of central server as well)

example: ZeroMQ:

- not a standard message broker but a lightweight messaging library a which provides messaging capabilities
- Provides **high-throughput and low-latency communication** by implementing **direct connection among producer and consumer**, with no intermediate entities involved
- It works as **a new layer on the networking stack**. It expands the concept of socket with a similar API but enhanced with built-in messaging patterns: Request/Reply, Publish/Subscribe, Pipeline and others
- In contrast with classic socket, each ZeroMQ socket comes with an internal queue to allow for asynchronous communication. This means that if the data is produced when the consumers is not running, the ZeroMQ library will take care of deferred delivery with no additional load on the producer side.
- mainly supports its own binary protocol
- provides limited messaging capabilities - some can be implemented easily in the application level (ack)
- Others are very hard to add: such as failover, multicast support for 1-N topology, guaranteed delivery, persistence

Message Systems - ZeroMQ

Ref: http://iopscience.iop.org/article/10.1088/1742-6596/608/1/012038/meta



it's best to think of ZeroMQ as networking library than an "MQ". It makes communication between processes easier than writing BSD sockets code by hand

Ref: https://news.ycombinator.com/item?id=9634801

HISTORY LESSON (101)

- BSD sockets begat *ZeroMQ*
- *ZeroMQ* begat *nanomsg*
- *nanomsg* begat *mangos*
- *mangos* begat *nng*

@Marina Popova

Messaging Systems: Options [too many ...]

@Marina Popova

Messaging Systems: Options

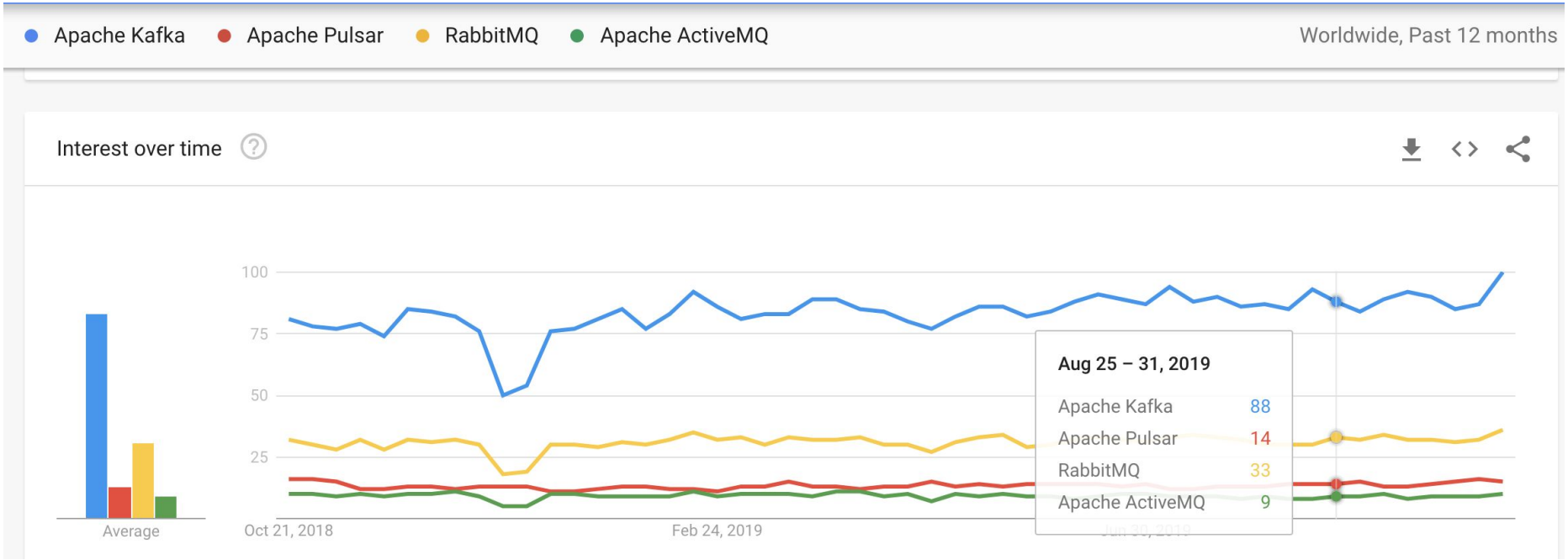
Apache Kafka
distributed pub-sub

Google Cloud Pub/Sub

Apache Pulsar
distributed pub-sub, point-to-point

Azure Message Storage

Messaging Systems: Options



Kafka

Apache Kafka is an open-source project originally from LinkedIn, now part of the Apache foundation.

Ref: https://www.tutorialspoint.com/apache_kafka/index.htm
https://kafka.apache.org/intro
http://cloudurable.com/blog/kafka-architecture/index.html

From Kafka official docs:

Kafka is designed for real-time, horizontally scalable, sub-millisecond-latency activity stream analytics, to move big amount of data from the producers to many potential consumers

The scale and the data size (billions of messages and hundreds of gigabytes per day) and latency requirements makes the use case not suitable for standard brokers

Kafka: main design decisions

- The crucial design decision of Kafka is to be a virtually **stateless broker**, and retain almost no information about consumers.
- A consumer has to retain its own state (e.g. the information about the last data read) and poll Kafka for new data when needed.
- This allows Kafka to **persist a single message copy independently from the number of consumers** (e.g. messages are not removed on consumption, but by retention period or other policy), with a resulting high-throughput for read and write operations

- Kafka persistence is implemented as a **distributed commit log**, designed as distributed system based on Zookeeper, which allows for unlimited horizontal scaling and automatic balancing of consumer/producer/broker players

- In contrast with standard message brokers, Kafka provides **limited messaging capabilities** (e.g. mainly topic semantic, file-system as unique persistent storage, strict guaranteed ordering).

Kafka

What is Kafka good for?

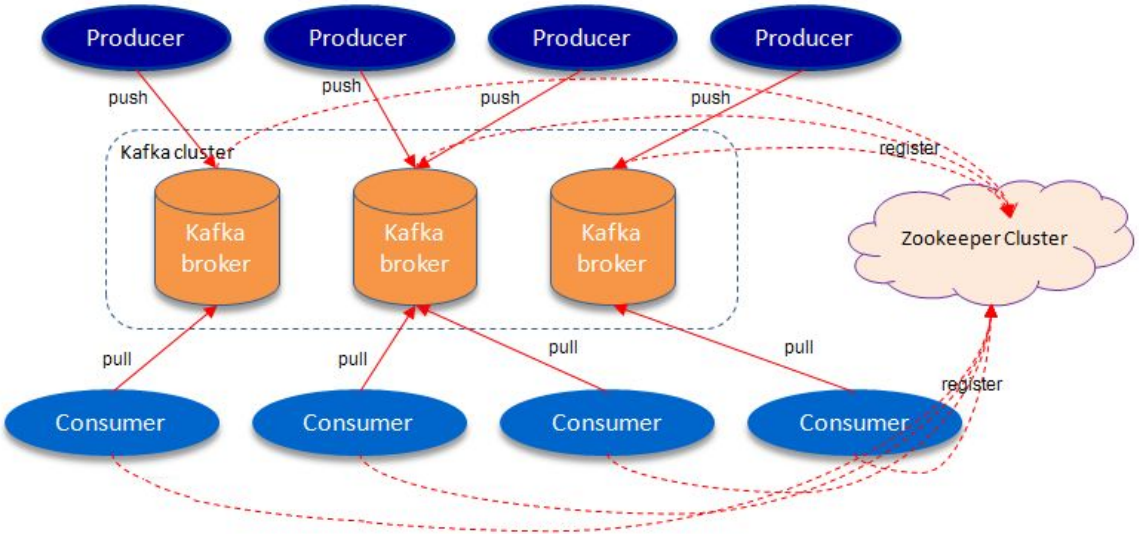
It gets used for two broad classes of application:

1. Building real-time streaming data pipelines that reliably get data between systems or applications
2. Building real-time streaming applications that transform or react to the streams of data

We will focus on the first class of functionality first - Kafka as a messaging platform
And we will get back to the second area, Kafka as a Streaming platform in the Lecture on Stream processing

Kafka - Main Concepts

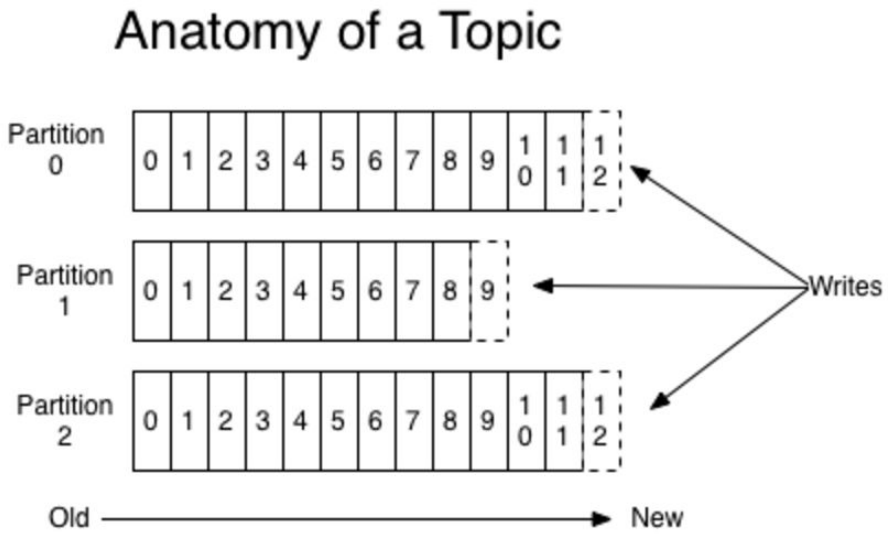
- Kafka is run as a cluster on one or more servers (nodes), called ***brokers***
- The Kafka cluster stores data as streams of ***records*** in categories called ***topics***
- Each record consists of a key, a value, and a timestamp - and is **immutable**
- Kafka ***producers*** write records to Topics
- Kafka ***consumers*** read records from Topics



https://medium.com/@mukeshkumar_46704/in-depth-kafka-message-queue-principles-of-high-reliability-42e464e66172

Kafka - Anatomy of Topics and Logs

- a ***topic*** is a category or feed name to which records are published
- A topic log consists of many ***partitions***
- a topic can have zero, one, or many consumers
- A topic is associated with a **log** - which is a data structure on disk
- Kafka appends records from a producer(s) to the end of a topic log
- each partition is an ordered, immutable sequence of records that is continually appended to—a structured commit log
- each record in the partitions is assigned a sequential id number called the ***offset*** that uniquely identifies each record within the partition



Kafka - Main Concepts

Logs and Partitions

- Partitions are spread over multiple files, which are called ***segments***
- partitions are replicated to many nodes - the number of copies is controlled by the ***replication factor***
- Kafka cluster **retains all published records** - whether or not they have been consumed - using a configurable ***retention period***

- This log design provides:
  a. Failover
  b. Horizontal scalability
  c. High performance of reads and writes
  d. 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.
  e. partitions act as the unit of parallelism

Kafka: Leaders, Followers, ISR

Partitions and Replication

- out of all servers that hold the replicas, one will be chosen as the ***leader* for this partition**, and others will be assigned as ***followers*** - this is done using Zookeeper

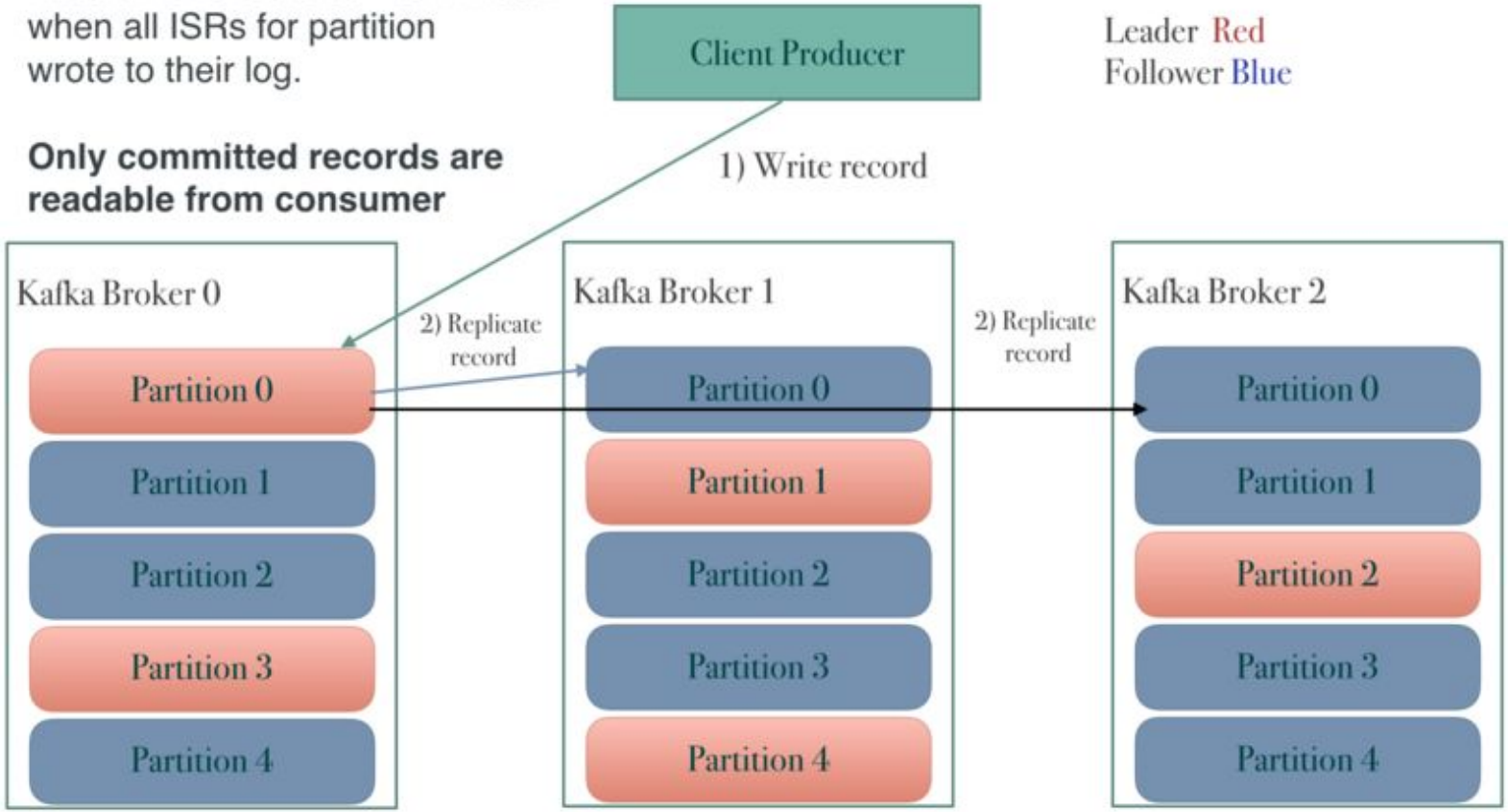
- each server handles its share of data and requests by sharing partition leadership
- the broker that has the partition leader h**andles all reads and writes** of records for the partition.
- writes to the leader partition are **replicated to followers**

- a follower that is in-sync is called an ***ISR (in-sync replica).***
- If a partition leader fails, Kafka chooses a new ISR as the new leader
- If a follower fails - a new node will be selected to be a follower and a copy of the partition created there
- the record write is considered “committed” when all ISRs for partition wrote to their log
- only committed records are readable from consumer.

Kafka Replication to Partition 0

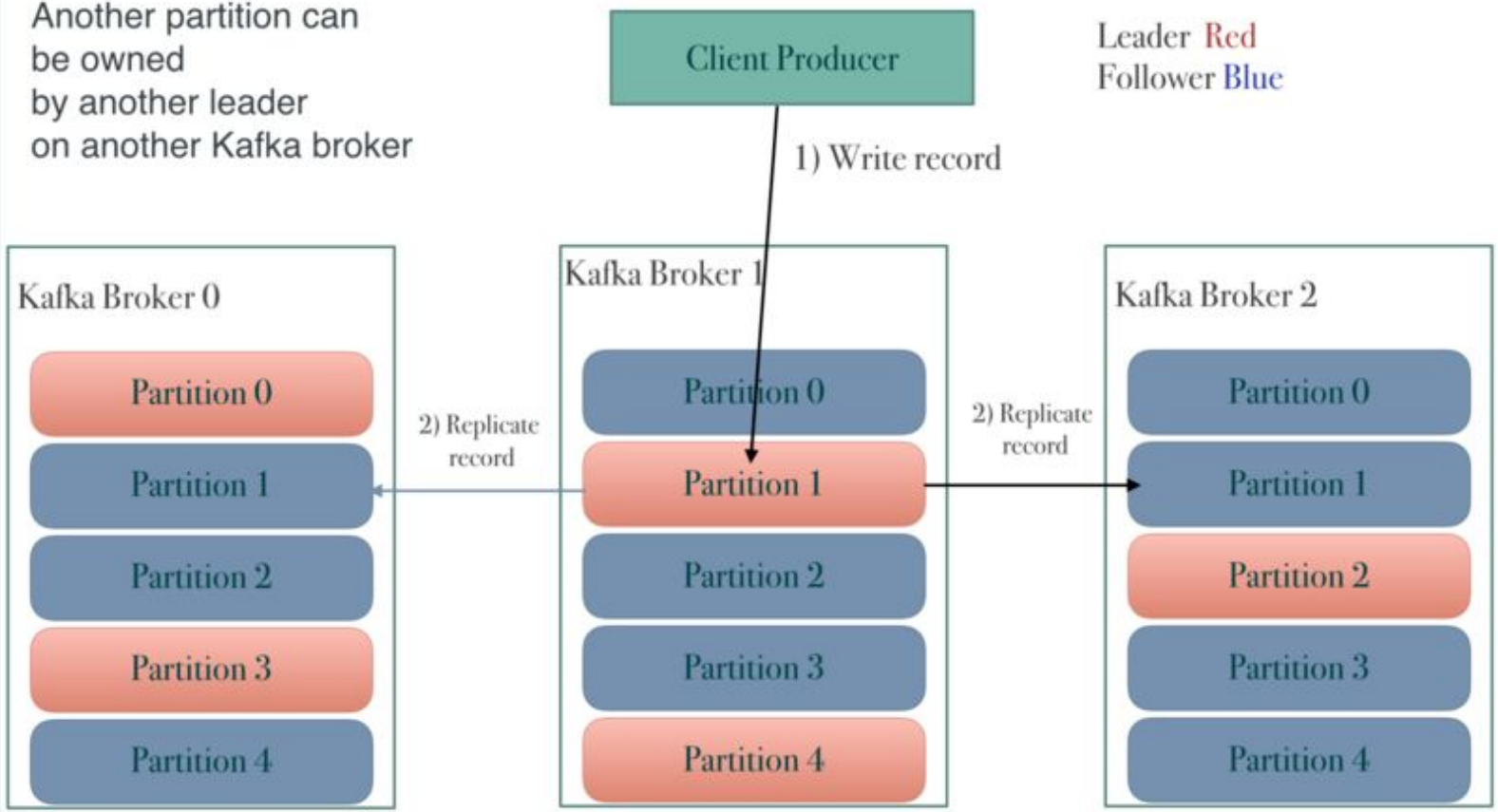
Record is considered "committed" when all ISRs for partition wrote to their log.

Only committed records are readable from consumer



Kafka Replication to Partitions 1

Another partition can
be owned
by another leader
on another Kafka broker



Kafka - Main Concepts

Example of topic/partition replication on a 5-node Kafka cluster:

```
=> /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic marina_test
Topic:marina_test       PartitionCount:10       ReplicationFactor:2     Configs:
        Topic: marina_test      Partition: 0    Leader: 5       Replicas: 5,4   Isr: 5,4
        Topic: marina_test      Partition: 1    Leader: 1       Replicas: 1,5   Isr: 1,5
        Topic: marina_test      Partition: 2    Leader: 2       Replicas: 2,1   Isr: 2,1
        Topic: marina_test      Partition: 3    Leader: 3       Replicas: 3,2   Isr: 3,2
        Topic: marina_test      Partition: 4    Leader: 4       Replicas: 4,3   Isr: 4,3
        Topic: marina_test      Partition: 5    Leader: 5       Replicas: 5,1   Isr: 5,1
        Topic: marina_test      Partition: 6    Leader: 1       Replicas: 1,2   Isr: 1,2
        Topic: marina_test      Partition: 7    Leader: 2       Replicas: 2,3   Isr: 2,3
        Topic: marina_test      Partition: 8    Leader: 3       Replicas: 3,4   Isr: 3,4
        Topic: marina_test      Partition: 9    Leader: 4       Replicas: 4,5   Isr: 4,5
```

Kafka - Main Concepts

Producers

- Kafka producers send records/messages to topics
- A record can consist of <key, value>, or it can contain <value> only
- The **producer picks which partition to send a record to,** per topic, based on the record's key, if it exists
- this functionality is performed by a **Partitioner**
- The default partitioner for Java uses a hash of the record's key to choose the partition or uses a round-robin strategy if the record has no key
- It is very easy to create and use a custom Partitioner, to partition messages by some other criteria

Kafka - Main Concepts

Message Durability

You can control the durability of messages written to Kafka through the **ACKS** setting.

- **ACKS = 1** - default value, requires an explicit acknowledgement from the partition leader that the write succeeded

- **ACKS = all** - the strongest guarantee that Kafka provides. It guarantees that not only did the partition leader accept the write, but it was **successfully replicated to all of the in-sync replicas**

- **ACKS = 0** - no response from broker is required. The weakest delivery guarantee. You can also use a value of “0” to maximize throughput, but you will have no guarantee that the message was successfully written to the broker’s log

Kafka - Main Concepts

Producers

Batching and Compression: Kafka producers attempt to collect sent messages into batches to improve throughput.

- use batch.size to control the maximum size in bytes of each message batch.
- To give more time for batches to fill, you can use linger.ms to have the producer delay sending.
- Compression can be enabled with the compression.type setting. Compression covers full message batches, so larger batches will typically mean a higher compression ratio.

Kafka distribution comes with a simple producer for testing, that can be run from a command line, it is called console producer:

./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic topic1

Kafka - Main Concepts

Consumers

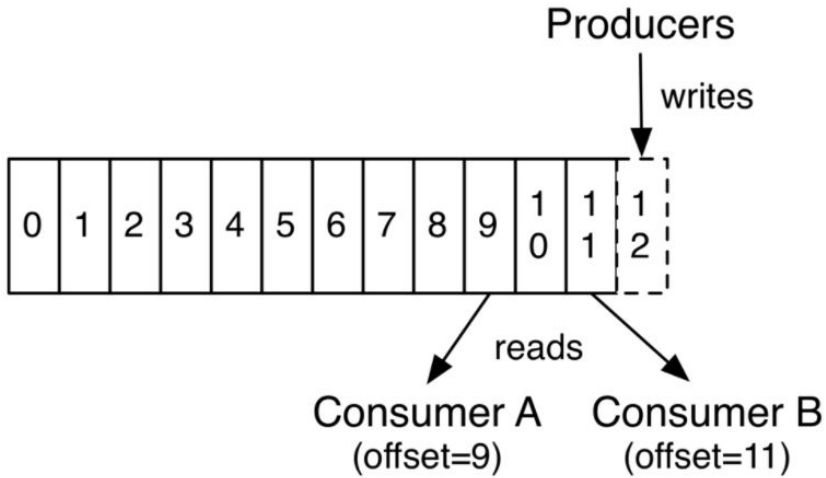
- Consumers read from Kafka topics at their own pace and Kafka keeps track of where each consumer is by storing, logically, a **<consumerID, partition, offset>** info for each topic and each consumer
- the only metadata retained on a per-consumer basis is the **offset or position of that consumer in the log**.
- This means that Kafka consumers are very cheap!!
- Kafka records committed offsets per consumer group in the special **"__conusmer_offset"** topic.

Kafka distribution comes with a simple console consumer that can be run from a command line and used for testing:
./bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic topic1

Kafka - more on offsets

the offset is controlled by the consumer:

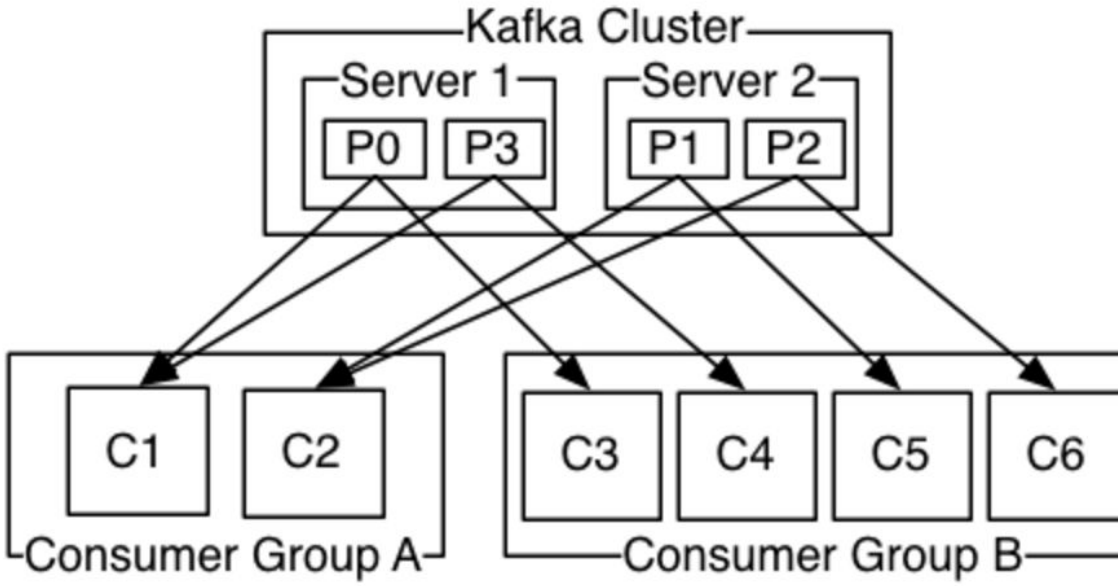
- consumer can commit offset itself, after it reads records - or at custom points in times, as programmed in the consumer logic
- alternatively, offsets can be committed by Kafka automatically if "auto-commit" is set to TRUE
- consumer can start reading records from any point in time:
  - it can reset to an "OLDEST" available offset to reprocess data from the past
  - can start from the "LATEST" offset to read the most recent record and start consuming from "now"
  - can reset offsets to some custom position in time



Kafka - Main Concepts

Consumers and Consumer Groups

- Consumers label themselves with a ***consumer group*** name
- "Consumer group" can have one or more "consumer instances"
- each record published to a topic is delivered to one consumer instance within each subscribing consumer group
- Each partition of a topic can be processed by **only one** consumer instance from a group
- Kafka divides partitions over consumer instances within a consumer group



Kafka - Main Concepts

Consumer parallelism and rebalancing

- Consumer membership within a consumer group is handled by the Kafka protocol dynamically
- If new consumer joins a consumer group, it gets a share of partitions.
- If a consumer dies, its partitions are split among the remaining live consumers in the consumer group. This is how Kafka does fail-over of consumers in a consumer group.
- Kafka only provides a total order over records *within* a partition, not between different partitions in a topic.

Kafka - Main Concepts

Kafka and Zookeeper

- Kafka uses ZooKeeper to manage the cluster.
- ZooKeeper is responsible for:
  a. coordinate of the brokers/cluster topology
  b. leadership election of Kafka Broker and Topic Partition pairs
  c. management of the service discovery for Kafka Brokers that form the cluster.
- Zookeeper sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc. Zookeeper provides an in-sync view of Kafka Cluster configuration.

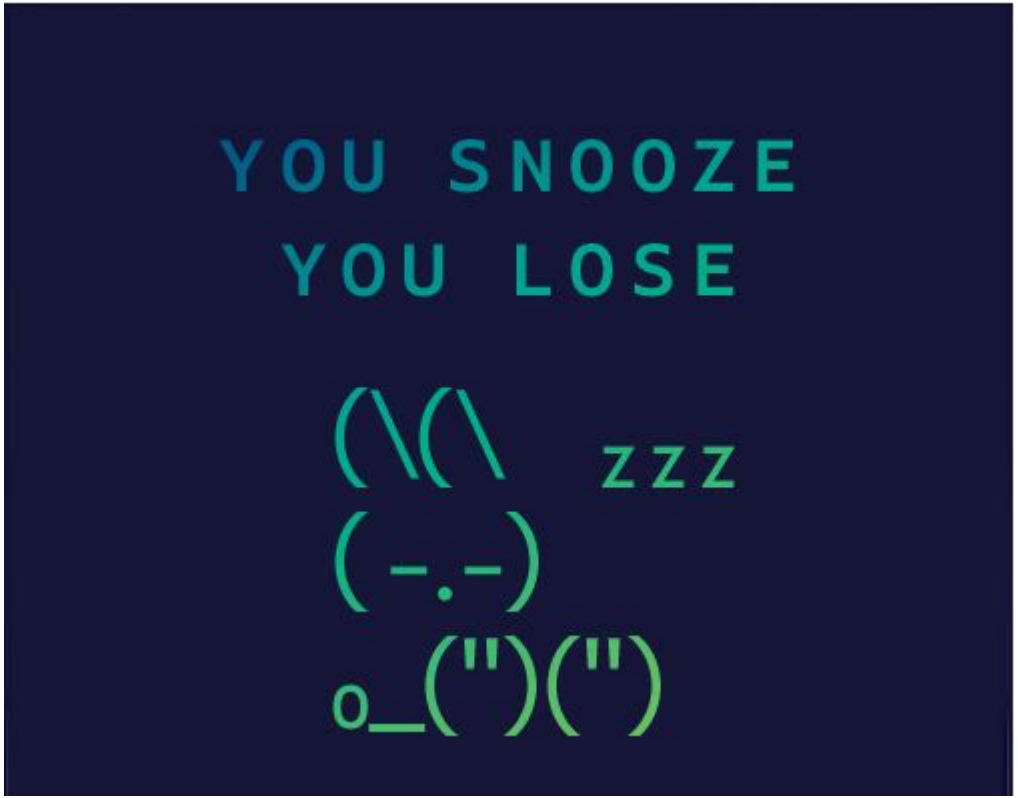
Kafka: Message Delivery semantics

How Kafka realizes the message delivery modes:

- "at-most-once" - Consumer reads message, saves offset, processes message
  - Problem: consumer process dies after saving the position but before processing the message - consumer takes over, starts at last position and the message is never processed
- “at-least-once" - Consumer reads message, process messages, saves offset
  - Problem: consumer could crash after processing message but before saving position - consumer takes over and receives already processed message
- “exactly once” - new Transaction mode added to Kafka since 0.10 - allows for exactly-once semantics at the [slight] expense of the performance degradation:
  - Ref: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

Kafka compared to MOMs

Slooooow Consumers …...



@Marina Popova

Kafka compared to MOMs

Slooooow Consumers …...

MOMs:

- For p2p: has to keep all data until the consumer is caught up (if ever) - the more message are persisted in the MOM, the faster its performance degrades
- For pub-sub:
  - The broker in a pub/sub system may be designed to deliver messages for a specified time, but then stop attempting delivery, whether or not it has received confirmation of successful receipt of the message by all subscribers.
  - If a confirmation of the receival is required - a pub/sub system designed this way may not be able to guarantee delivery of messages
  - Tighter coupling of the designs of such a publisher and subscriber pair must be enforced outside of the pub/sub architecture to accomplish such assured delivery (e.g. by requiring the subscriber to publish receipt messages).
  - A publisher in a pub/sub system may assume that a subscriber is listening, when in fact it is not.

Kafka compared to MOMs

The pub/sub pattern scales well for small networks with a small number of publisher and subscriber nodes and low message volume. However, as the number of nodes and messages grows, the likelihood of instabilities increases, limiting the maximum scalability of a pub/sub network. Example throughput instabilities at large scales include:

- Load surges—periods when subscriber requests saturate network throughput followed by periods of low message volume (underutilized network bandwidth)
- Slowdowns—as more and more applications use the system (even if they are communicating on separate pub/sub channels) the message volume flow to an individual subscriber will slow

Kafka:

does not care if consumers are slow or alive at all - any consumer can start getting data from any point in time (offset), as long as the data is not removed due to the retention period

Kafka compared to MOMs

Major Differences between Kafka and traditional MOMs
Smart Broker - Dumb Consumer vs Dumb Broker - Smart Consumers

MOM: Smart Broker - Dumb Consumer model:

- Broker keeps track of deliveries of messages to all consumers
- Broker is responsible for re-delivery to guarantee no message loss
- Broker is responsible for the complex life cycle of events in the queues - events are usually removed from the queues when all consumers retrieved them

Kafka: Dumb Broker - Smart Consumers model:

- Broker keeps ALL messages in a reliable manner, for configured amount of time (retention period)
- Broker only keeps offsets (point to a location in the log file) for each consumer - thus, hundreds and thousands of consumers can co-exist and read the data at their own speed
- Consumers are responsible for recovering themselves from failures - and can choose to re-process events from a known failure point

Kafka - configuration

Most important configuration for one node cluster setup:

server.properties:

```
# The id of the broker. This must be set to a unique integer for each broker.
broker.id=0
# Switch to enable topic deletion or not, default value is false
delete.topic.enable=true
log.dirs=/Users/marinapopova/Marina/data/kafka-logs-2.0
zookeeper.connect=localhost:2181
```

zookeeper.properties:

```
dataDir=/Users/marinapopova/Marina/data/zookeeper-2.0
```

Kafka - important commands

Start the ZK server:
./bin/zookeeper-server-start.sh ./config/zookeeper.properties

Start Kafka server:
./bin/kafka-server-start.sh ./config/server.properties

Create a topic and list all topics:
./bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 2 --topic topic1
./bin/kafka-topics.sh --list --zookeeper localhost:2181

Run console consumer and producer, inspect groups:
./bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic topic1
./bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topic1
./bin/kafka-consumer-groups.sh --describe --group group_m1 --bootstrap-server localhost:9092

Inspect Logs:
./bin/kafka-run-class.sh kafka.tools.DumpLogSegments --files
~/Marina/data/kafka-logs-2.0/topic1-0/00000000000000000000.log --print-data-log

Kafka - Main APIs

Producer APIs

https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html

Class KafkaProducer<K,V>

Main methods:

`Future<RecordMetadata>`	`send(ProducerRecord<K,V> record)` Asynchronously send a record to a topic.

Class ProducerRecord<K,V>

```
public class ProducerRecord<K,V>
extends Object
```

A key/value pair to be sent to Kafka. This consists of a topic name to which the record is being sent, an optional partition number, and an optional key and value.

If a valid partition number is specified that partition will be used when sending the record. If no partition is specified but a key is present a partition will be chosen using a hash of the key. If n
key nor partition is present a partition will be assigned in a round-robin fashion.

The record also has an associated timestamp. If the user did not provide a timestamp, the producer will stamp the record with its current time. The timestamp eventually used by Kafka depe
the timestamp type configured for the topic.

Kafka - Main APIs

Consumer APIs

Class KafkaConsumer<K,V>

Main methods:

void	**subscribe**(**Collection**<**String**> topics) Subscribe to the given list of topics to get dynamically assigned partitions.

ConsumerRecords<**K**,**V**>	**poll**(long timeout) Fetch data for the topics or partitions specified using one of the subscribe/assign APIs.

Kafka - Main APIs

Kafka APIs - more docs and examples in other languages:

https://docs.confluent.io/current/clients/index.html

https://docs.confluent.io/current/clients/producer.html
https://docs.confluent.io/current/clients/consumer.html