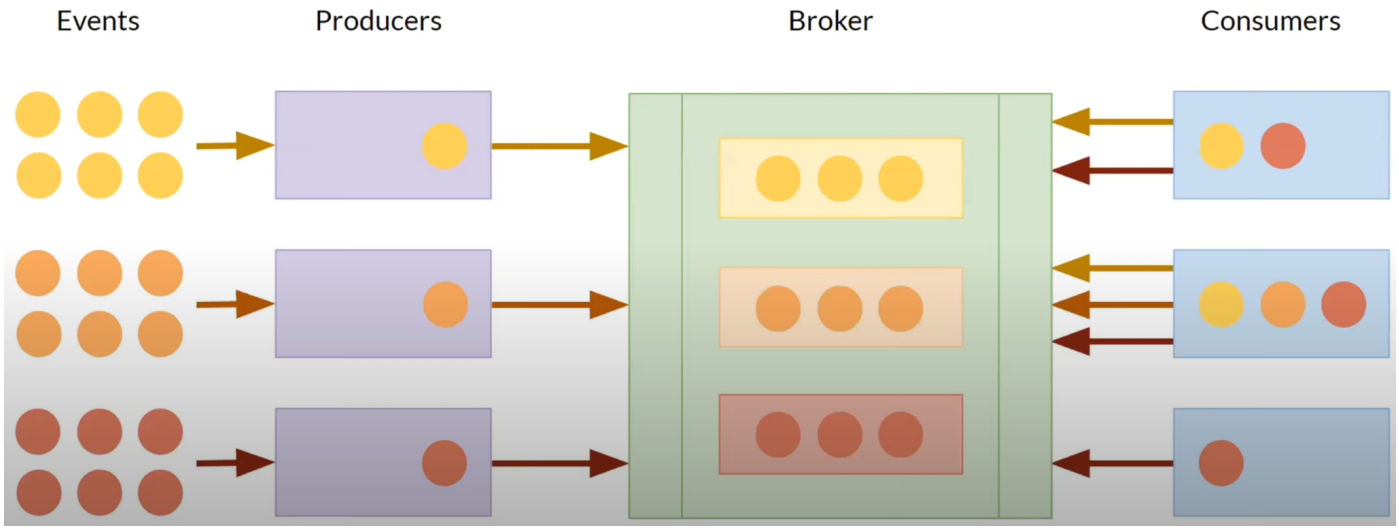
Kafka

https://www.youtube.com/watch?v=-AZOi3kP9Js

https://www.youtube.com/watch?v=BtmYjTO1EpI

Kafka is an open-source **message broker** project developed by the Apache Software Foundation written in Scala and is a distributed **publish-subscribe messaging system**. Kafka's design pattern is mainly based on the **transactional logs** design.



What can you do with Kafka?

- In order to transmit data between two systems, we can build a real-time stream of data pipelines with it.
- Also, we can build a real-time streaming platform with Kafka, that can actually react to the data.

Feature	**Description**
High Throughput	Support for millions of messages with modest hardware
Scalability	Highly scalable distributed systems with no downtime
Replication	Messages are replicated across the cluster to provide support for multiple subscribers and balances the consumers in case of failures
Durability	Provides support for persistence of message to disk
Stream Processing	Used with real-time streaming applications like Apache Spark & Storm
Data Loss	Kafka with proper configurations can ensure zero data loss

List the various components in Kafka:

- Topic – a stream of messages belonging to the same type
- Producer – that can publish messages to a topic
- Brokers – a set of servers where the publishes messages are stored
- Consumer – that subscribes to various topics and pulls data from the brokers.

Zookeeper

Kafka uses **Zookeeper** to store offsets of messages consumed for a specific topic and partition by a specific Consumer Group. (Message offset role: messages contained in the partitions are assigned a unique ID number that is called the offset. The role of the offset is to uniquely identify every message within the partition)

It is not possible to bypass Zookeeper and connect directly to the Kafka server. If, for some reason, ZooKeeper is down, you cannot service any client request.

Apache ZooKeeper is an open-source server for highly reliable distributed coordination of cloud applications. It is a project of the Apache Software Foundation.

ZooKeeper is essentially a service for distributed systems offering a hierarchical **key-value store**, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. ZooKeeper was a sub-project of Hadoop but is now a top-level Apache project in its own right.

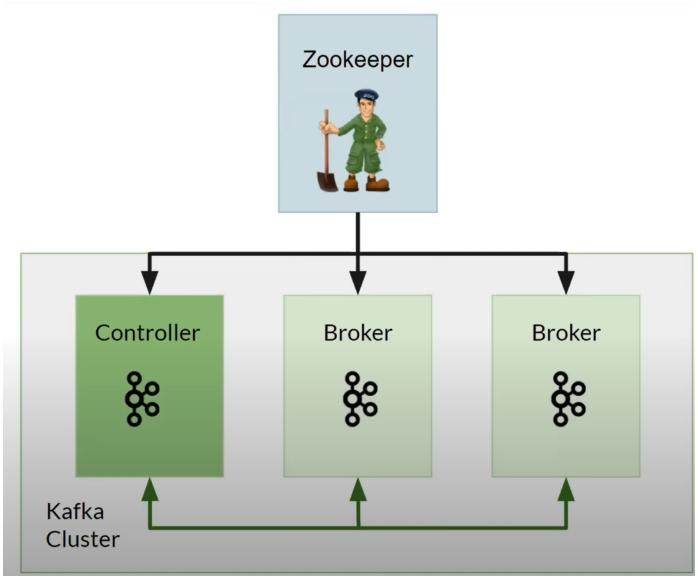
Typical use cases for ZooKeeper are:

- Naming service
- Configuration management
- Data Synchronization
- Leader election
- Message queue
- Notification system

Kafka message contains:

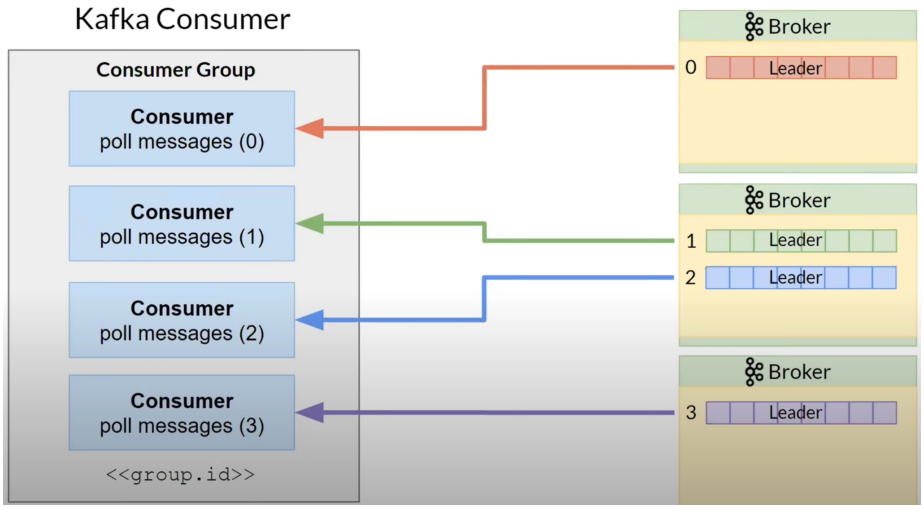
- key (optional) - used for message distribution across partitions
- value - byte array
- timestamp - set either before sending by clien or after receiving by broker
- headers - key-value pairs/attributes

Kafka Cluster



We may want to process messages with multiple consumers. We can organize concurrent processing of a topic using multiple partitions, and each partition is attached to its consumers using a group mechanism, as shown below.

- Single consumer can process more than 1 partition in a topic when **NoPartitions > NoConsumers**.
- Some consumers stay idle (have no job to do) when **NoPartitions < NoConsumers**

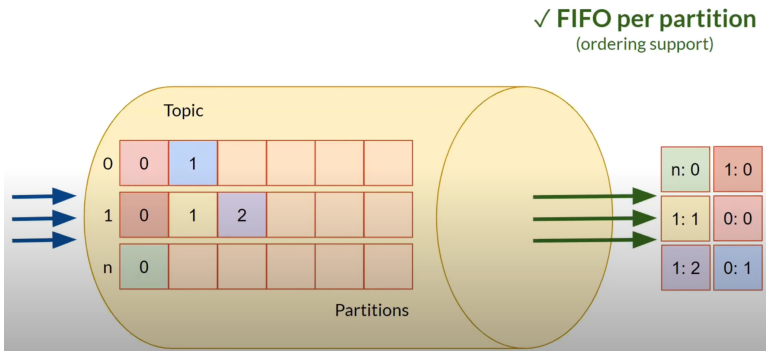


In order to overcome the challenges of collecting the large volume of data, and analyzing the collected data we need a messaging system. Hence Apache Kafka came into the story. Its benefits are:

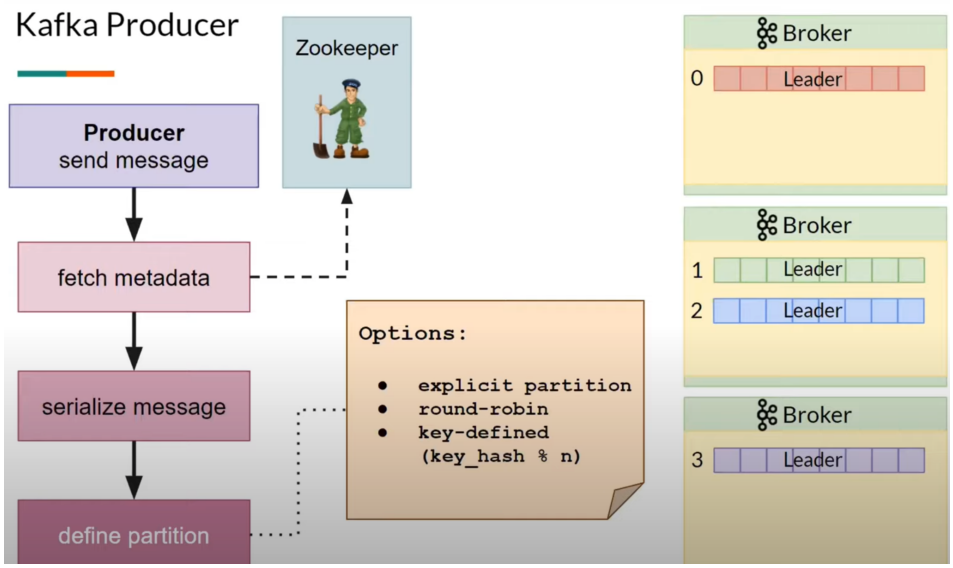
- It is possible to track web activities just by storing/sending the events for real-time processes.
- Through this, we can Alert as well as report the operational metrics.
- Also, we can transform data into the standard format.

- Moreover, it allows continuous processing of streaming data to the topics.

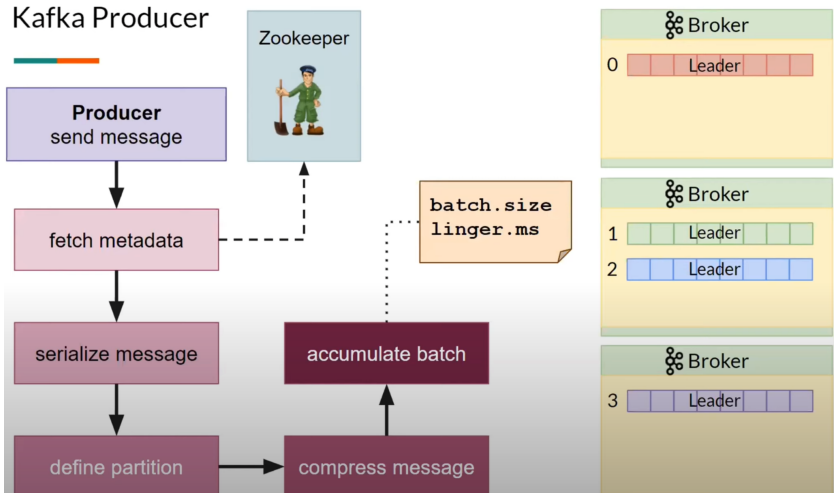
Topics and partitions



Choosing partition to send the message to:



Sending batch to broker:



The optimal number of partitions (for maximum throughput) per cluster is around the number of CPU cores (or slightly more, up to 100 partitions), i.e.cluster **CPU cores >= optimal partitions <= 100**

Too many partitions result in a significant drop in throughput (however, you can get increased throughput for more partitions by increasing the size of your cluster).

Setting producer **acks=all can give comparable or even slightly better throughput** compared with the default of acks=1.

Setting producer **acks=all results in higher latencies** compared with the default of acks=1.

Both producer acks=all and idempotence=true have comparable durability, throughput, and latency (i.e. the only practical difference is that **idempotence=true guarantees exactly-once semantics for producers**).

Topic VS Queue

Is Kafka a Topic or a Queue?

https://abhishek1987.medium.com/kafka-is-it-a-topic-or-a-queue-30c85386afd6

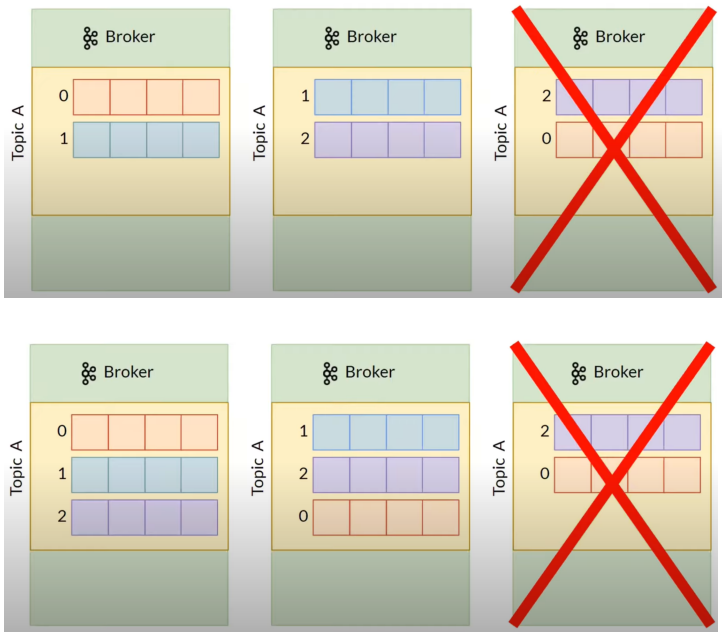
Kafka is both a Topic and a Queue. Queue based systems are typically designed in a way that there are multiple consumers processing data from a queue and the work gets distributed such that **each consumer gets a different set of items to process**. Hence there is no overlap, allowing the workload to be shared and enables horizontally scalable architectures.

A Kafka topic is sub-divided into units called partitions for fault tolerance and scalability. **Consumer Groups allow Kafka to behave like a Queue**, since each consumer instance in a group processes data from a non-overlapping set of partitions (within a Kafka topic).

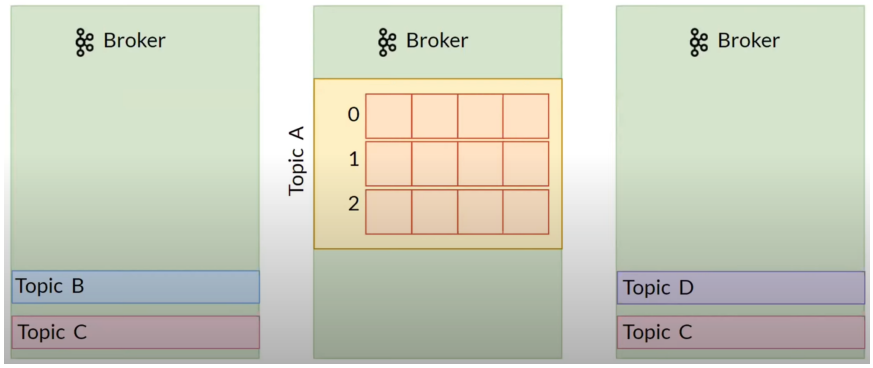
Replication

Replicas are essentially a list of nodes that **replicate the log for a particular partition** irrespective of whether they play the role of the Leader. On the other hand, ISR stands for In-Sync Replicas. It is essentially a set of message replicas that are synced to the leaders.

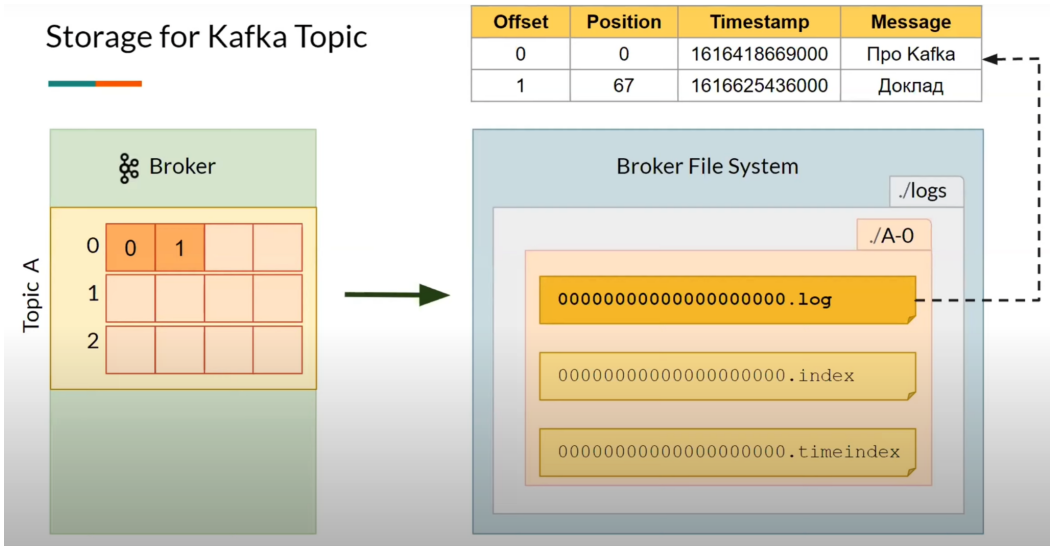
Replication ensures that published messages are not lost and can be consumed in the event of any machine error, program error or frequent software upgrades.

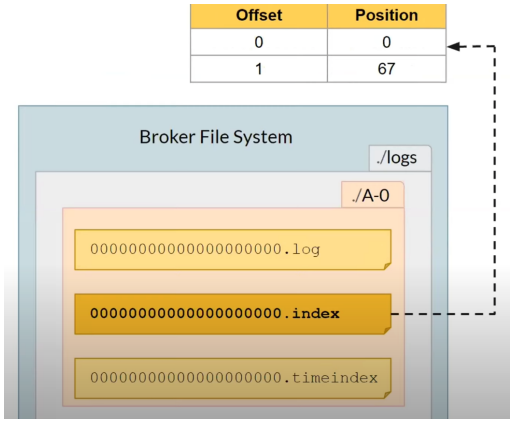


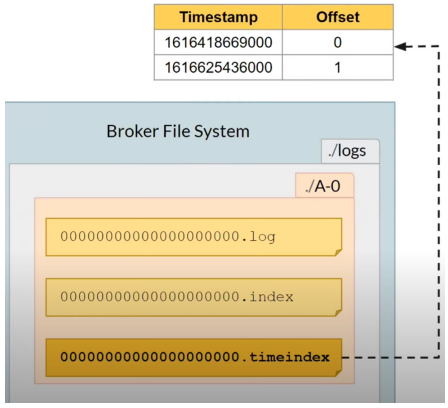
Problem of imbalance partition placement be like:



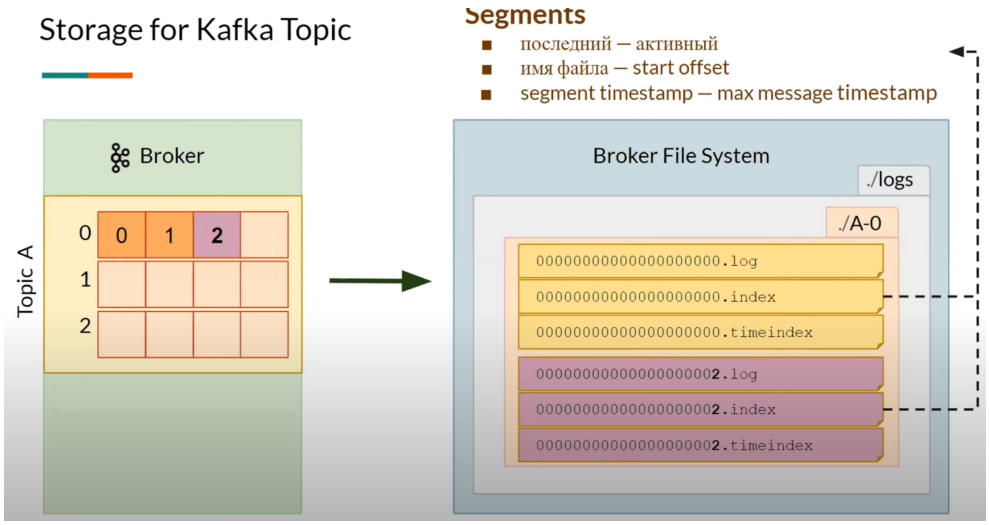
Data is stored in log files:


Storage for Kafka Topic
Offset
Position
Timestamp
Message
0
0
1616418669000
Про Kafka
1
67
1616625436000
Доклад
Broker
Topic A
0
1
2
0
1
Broker File System
./logs
./A-0
00000000000000000000.log
00000000000000000000.index
00000000000000000000.timeindex


Offset
Position
0
0
1
67
Broker File System
./logs
./A-0
00000000000000000000.log
00000000000000000000.index
00000000000000000000.timeindex

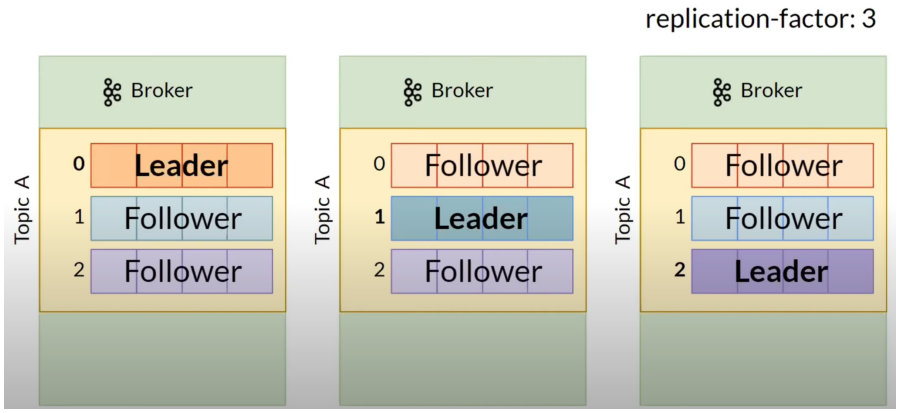

Timestamp
Offset
1616418669000
0
1616625436000
1
Broker File System
./logs
./A-0
00000000000000000000.log
00000000000000000000.index
00000000000000000000.timeindex

Every .log file size is limited:

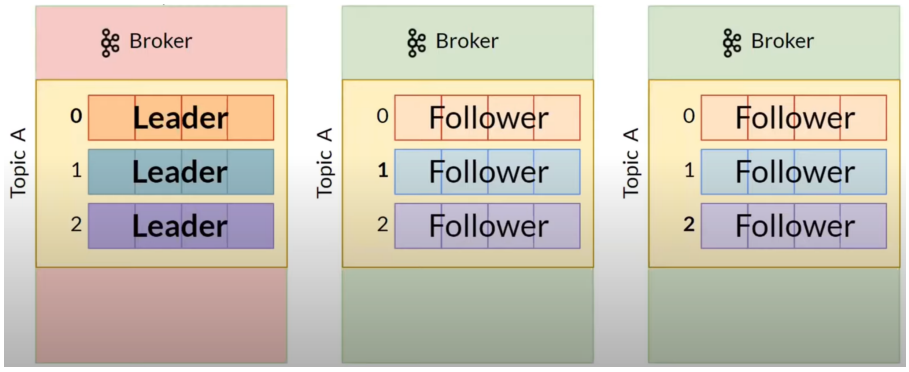


Leader-Follower

Every partition in Kafka has one server which plays the role of a Leader, and none or more servers that act as Followers. **The Leader performs the task of all read and write requests** for the partition, while the role of the Followers is to passively replicate the leader. In the event of the Leader failing, one of the Followers will take on the role of the Leader. This ensures load balancing of the server.

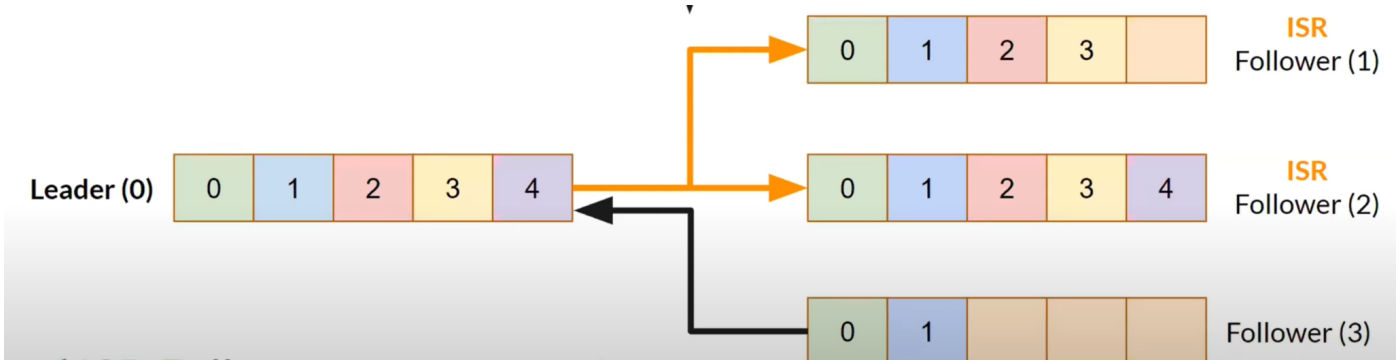


Imbalance leader election problem (Broker #1 is under heavy load)



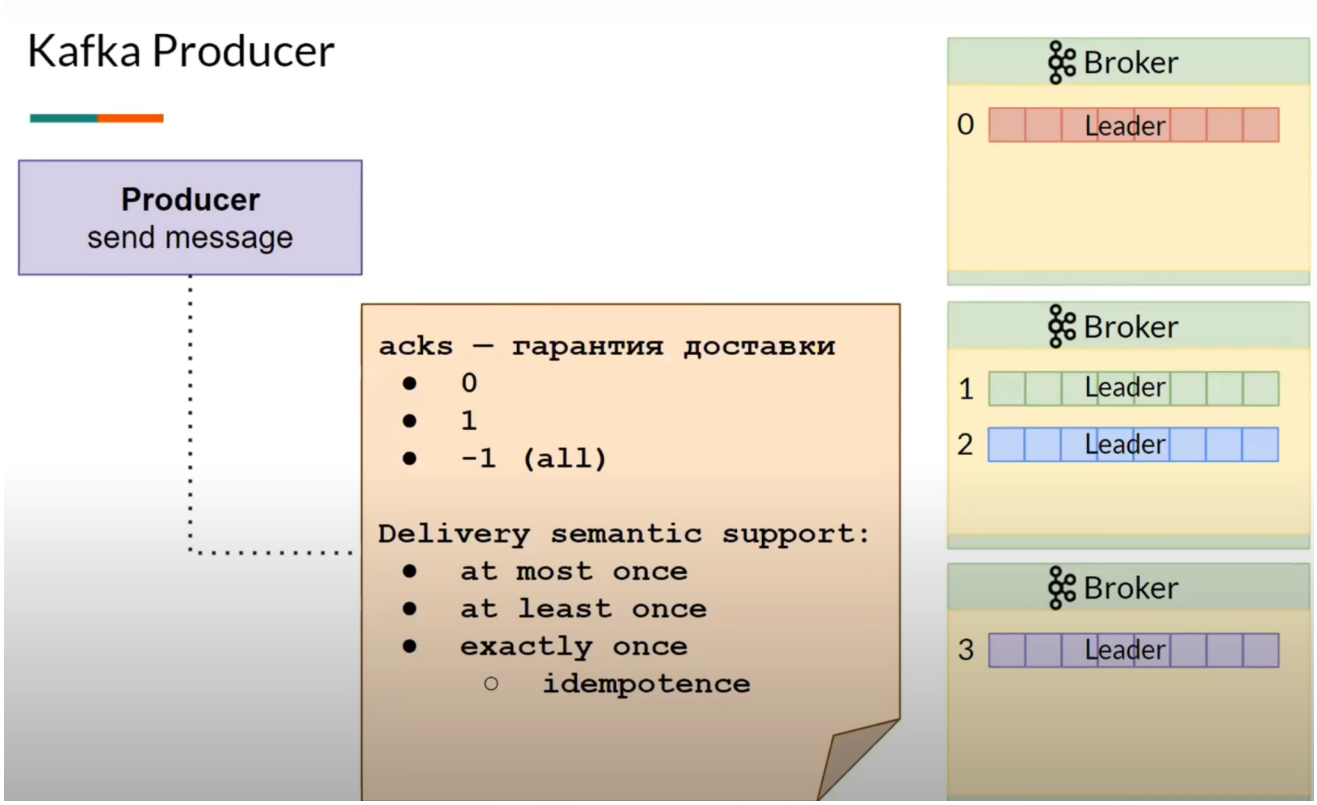
Data loss

A data loss can happen during leader reelection even if you have replication factor > 1, because generally speaking replicas are out-of-sync with the leader. In order to solve this problem, kafka supports In-Sync-Replicas (ISR).
Leader makes sure all the ISR replicas are in-sync before sending ‘ack’-response to the producer. ISRs are good candidates for a leader reelection.

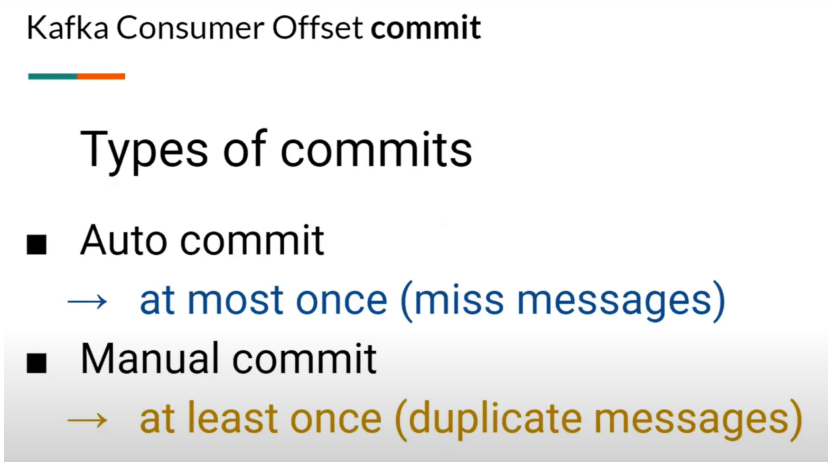


Delivery semantics

ack = 0 - no confirmation from broker required
ack = 1 - leader has processed the message
ack = -1 (all) - leader and all ISR have processed the message



What guarantees of delivery/message processing do we have?

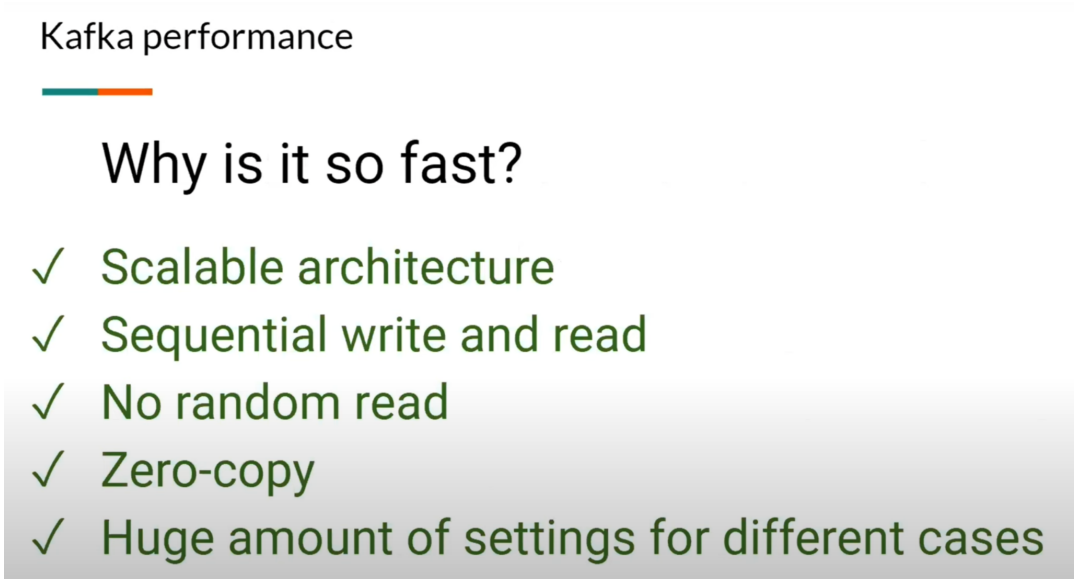


At most once scenario (auto commit):
Consumer pulls message -> auto commit happens (update an offset for a group in __consumer_offset topic) -> consumer fails -> ~~consumer handles the message~~.
The offset is updated but the message has not been processed. **The message is lost**.

At least once scenario (manual commit):
Consumer pulls message -> consumer handles the message -> consumer fails -> ~~consumer manually commits~~.
After the consumer restarts the same message would be pulled and processed all over again. **Duplicate processing**.

Exactly once scenario:
Implement a mechanism of storing offset in an external storage, do not rely on __consumer_offset data.
Example:
Event X = money transfer from User A to User B. We want to handle this event exactly once, obviously.
Suppose we use RDBMS that supports transactions. In that DB, we store info about user accounts as well as the kafka offset value for the most recent operation processed. After reading Event X, we perform a money transfer and update the offset value within a single transaction.

Performance



Starting kafka

Typical docker-compose.yml running zookeeper + kafka:

```
version: '2'
services:
  zookeeper:
    container_name: zookeeper
    networks:
      - kafka_network
    ...

  kafka:
    container_name: kafka
    networks:
      - kafka_network
    ports:
      - 29092:29092
    environment:
      KAFKA_LISTENERS: EXTERNAL_SAME_HOST://:29092,INTERNAL://:9092
      KAFKA_ADVERTISED_LISTENERS:
INTERNAL://kafka:9092,EXTERNAL_SAME_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
INTERNAL:PLAINTEXT,EXTERNAL_SAME_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
     ...

networks:
  kafka_network:
    name: kafka_docker_example_net
```

Kafka Listeners explained

https://rmoff.net/2018/08/02/kafka-listeners-explained/

and even better article:

https://www.confluent.io/blog/kafka-client-cannot-connect-to-broker-on-aws-on-docker-etc/

Example config:

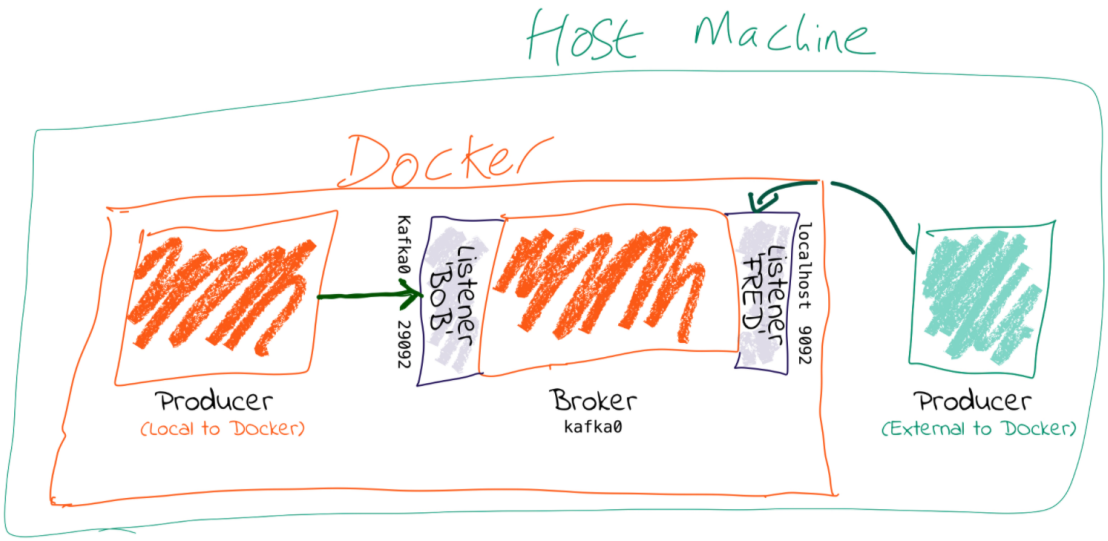
```
KAFKA_LISTENERS: LISTENER_BOB://kafka0:29092,LISTENER_FRED://localhost:9092

KAFKA_ADVERTISED_LISTENERS: LISTENER_BOB://kafka0:29092,LISTENER_FRED://localhost:9092

KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: LISTENER_BOB:PLAINTEXT,LISTENER_FRED:PLAINTEXT

KAFKA_INTER_BROKER_LISTENER_NAME: LISTENER_BOB
```

tl;dr : You need to set advertised.listeners (or KAFKA_ADVERTISED_LISTENERS if you're using Docker images) to the external address (host/IP) so that clients can correctly connect to it. Otherwise they'll try to connect to the internal host address–and if that's not reachable then problems ensue.



You need to tell Kafka how the brokers can reach each other, but also make sure that external clients (producers/consumers) can reach the broker they need to.

The key thing is that when you run a client, the broker you pass to it is just where it's going to go and get the metadata about brokers in the cluster from. The actual host & IP that it will connect to for reading/writing data is based on **the data that the broker passes back in that initial connection**—even if it's just a single node and the broker returned is the same as the one connected to.

- KAFKA_LISTENERS is a comma-separated list of listeners, and the host/ip and port to which Kafka binds to on which to listen. For more complex networking this might be an IP address associated with a given network interface on a machine. The default is 0.0.0.0, which means listening on all interfaces. (listeners)
- KAFKA_ADVERTISED_LISTENERS is a comma-separated list of listeners with their host/ip and port. This is the metadata that's passed back to clients. (advertised.listeners)
- KAFKA_LISTENER_SECURITY_PROTOCOL_MAP defines key/value pairs for the security protocol to use, per listener name. (listener.security.protocol.map)
- KAFKA_INTER_BROKER_LISTENER_NAME: Kafka brokers communicate between themselves, usually on the internal network (e.g. Docker network, AWS VPC, etc). To define which listener to use, specify KAFKA_INTER_BROKER_LISTENER_NAME (inter.broker.listener.name). The host/IP used must be accessible from the broker machine to others.

Configuration example

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.ByteArrayDeserializer
      auto-offset-reset: earliest
      group-id: "group-1"
      enable-auto-commit: false
    producer:
      acks: 1
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.ByteArraySerializer
      compression-type: gzip
```

Compression type should be set for both broker (or at a topic level) and producer. It is generally advised to set compression-type:producer on the broker side (for a topic) so the broker uses compression type set by producer.

```
//creating topic programmatically with KafkaAdmin&AdminClient
@Bean
 NewTopic myTopic() {
        var props = new HashMap<String, String>();
        props.put("compression.type", "producer");
        NewTopic newTopic = new NewTopic(brokerProperties.getTopicName(), 1, (short) 1);
        newTopic.configs(props);
        return newTopic;
}
```

Decompression is done implicitly on the consumer side, so we always see a decompressed message, no additional configuration required.

Producer batch configs:
linger.ms (default is 0)
batch.size (default is 16384 bytes)

After the Kafka producer collects a **batch.size** worth of messages it will send that batch. But, Kafka waits for **linger.ms** amount of milliseconds. Since linger.ms is 0 by default, Kafka won't batch messages and send each message immediately.

Useful commands

Create topic
kafka-topics --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic test-topic --config compression.type=(gzip|snappy|lz4|zstd)

Send to topic
kafka-console-producer --bootstrap-server localhost:9092 --topic test-topic < ~/test-1.file

Modify existing topic config:
kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name test-topic --alter --add-config compression.type=gzip

List log files and partitions (+ its size) for the topics:
kafka-log-dirs --describe --bootstrap-server 127.0.0.1:9092 --topic-list payroll-chunks

```
{"version":1,"brokers":[{"broker":1,"logDirs":[{"logDir":"/var/lib/kafka/data","error":null,"partitions":[{"partition":"payroll-chunks-0","size":235,"offsetLag":0,"isFuture":false}]}]}]}
```

List all the topics:
root@dcf52f5bbab6:/bin# **kafka-topics --bootstrap-server localhost:9092 --list**

```
__confluent.support.metrics
__consumer_offsets
large-file-chunks-topic
```

Describe topic configs:
kafka-topics --describe --zookeeper localhost:2181 --topic payroll-chunks

```
Topic:payroll-chunks    PartitionCount:1        ReplicationFactor:1
Configs:compression.type=producer,max.message.bytes=15728640
        Topic: payroll-chunks   Partition: 0    Leader: 1       Replicas: 1     Isr: 1
```

Check how the records are encrypted in a topic:
kafka-run-class kafka.tools.DumpLogSegments --files /var/lib/kafka/data/payroll-chunks-0/00000000000000000000.log --print-data-log | grep compresscodec

```
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 0 CreateTime: 1636802822601 size: 2018164 magic: 2 compresscodec: LZ4 crc: 972834431 isvalid: true
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 2018164 CreateTime: 1636802852084 size: 2018164 magic: 2 compresscodec: LZ4 crc: 780224980 isvalid: true
baseOffset: 2 lastOffset: 2 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 4036328 CreateTime: 1636802853851 size: 2018164 magic: 2 compresscodec: LZ4 crc: 1970859284 isvalid: true
baseOffset: 3 lastOffset: 3 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 6054492 CreateTime: 1636802866257 size: 2018164 magic: 2 compresscodec: LZ4 crc: 1087921445 isvalid: true
```

Large messages

The message max size is 1MB (the setting in your brokers is called message. max. bytes) Apache Kafka. If you really needed it badly, you could increase that size and make sure to increase the network buffers for your producers and consumers.

Step №1: Broker/topic configuration
We could allow all topics on a Broker to accept messages of greater than 1MB in size by setting the broker property:
message.max.bytes=20971520 (env prop for cp-kafka: KAFKA_MESSAGE_MAX_BYTES)

But it is better to set a limit per topic (at topic-level):
./kafka-topics.sh --bootstrap-server localhost:9092 --create --topic longMessage --partitions 1 \
--replication-factor 1 --config **max.message.bytes**=20971520

Additional property for replication (not mandatory)

replica.fetch.max.bytes

The number of bytes of messages to attempt to fetch for each partition. This is not an absolute maximum, if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that progress can be made. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config).

Type:	int
Default:	1048576 (1 mebibyte)
Valid Values:	[0,...]
Importance:	medium
Update Mode:	read-only

Step №2: Producer configuration
The property “**max.request.size**” needs to be set for producer configuration (ProducerConfig.MAX_REQUEST_SIZE_CONFIG in the Kafka Client library):

max.request.size

The maximum size of a request in bytes. This setting will limit the number of record batches the producer will send in a single request to avoid sending huge requests. This is also effectively a cap on the maximum uncompressed record batch size. Note that the server has its own cap on the record batch size (after compression if compression is enabled) which may be different from this.

Type:	int
Default:	1048576
Valid Values:	[0,...]
Importance:	medium

This setting controls the size of a produce request sent by the producer. It caps both the size of the largest message that can be sent and the number of messages that the producer can send in one request. For

example, with a default maximum request size of 1 MB, the largest message you can send is 1MB or the producer can batch 1000 messages of size 1k each into one request.

```
public ProducerFactory<String, String> producerFactory() {
    Map<String, Object> configProps = new HashMap<>();
    configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
    configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
    configProps.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG, "20971520");

    return new DefaultKafkaProducerFactory<>(configProps);
}
```

Or using application properties (applied in KafkaAutoConfiguration):

```
spring.kafka:
        producer:
            key-serializer: org.apache.kafka.common.serialization.StringSerializer
            value-serializer: org.apache.kafka.common.serialization.ByteArraySerializer
            …
            properties:
                max-request-size: 15728640
```

Step №3: Consumer configuration
Consumer configuration isn't mandatory for consuming large messages, avoiding them can have a performance impact on the consumer application.

max.partition.fetch.bytes

The maximum amount of data per-partition the server will return. Records are fetched in batches by the consumer. If the first record batch in the first non-empty partition of the fetch is larger than this limit, the batch will still be returned to ensure that the consumer can make progress. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config). See fetch.max.bytes for limiting the consumer request size.

Type: int
Default: 1048576 (1 mebibyte)
Valid Values: [0,...]
Importance: high

fetch.max.bytes

The maximum amount of data the server should return for a fetch request. Records are fetched in batches by the consumer, and if the first record batch in the first non-empty partition of the fetch is larger than this value, the record batch will still be returned to ensure that the consumer can make progress. As such, this is not a absolute maximum. The maximum record batch size accepted by the broker is defined via `message.max.bytes` (broker config) or `max.message.bytes` (topic config). Note that the consumer performs multiple fetches in parallel.

Type: int
Default: 52428800 (50 mebibytes)
Valid Values: [0,...]
Importance: medium

li-apache-kafka-clients

https://github.com/linkedin/li-apache-kafka-clients
li-apache-kafka-clients is a wrapper Kafka clients library built on top of vanilla Apache Kafka clients.

Large message support
Like many other messaging systems, Kafka has a maximum message size limit for a few reasons (e.g. memory management. Due to the message size limit, in some cases, user have to bear with small amount of message loss or involve external storage to store the large objects. li-apache-kafka-clients addresses this problem with a **solution that does not require an external storage dependency**. For users who are storing offsets checkpoints in Kafka, li-apache-kafka-clients handles large messages almost transparently.

large.message.enabled
max.message.segment.bytes
segment.serializer

If **large.message.enabled=**true, LiKafkaProducerImpl will split the messages whose serialized size is greater than **max.message.segment.bytes** into multiple LargeMessageSegment, serialize each LargeMessageSegment with the specified **segment.serializer** and send the serialized segments to Kafka brokers.