Kafka Broker:

Java Version recommend is java 1.8 with G1 GC

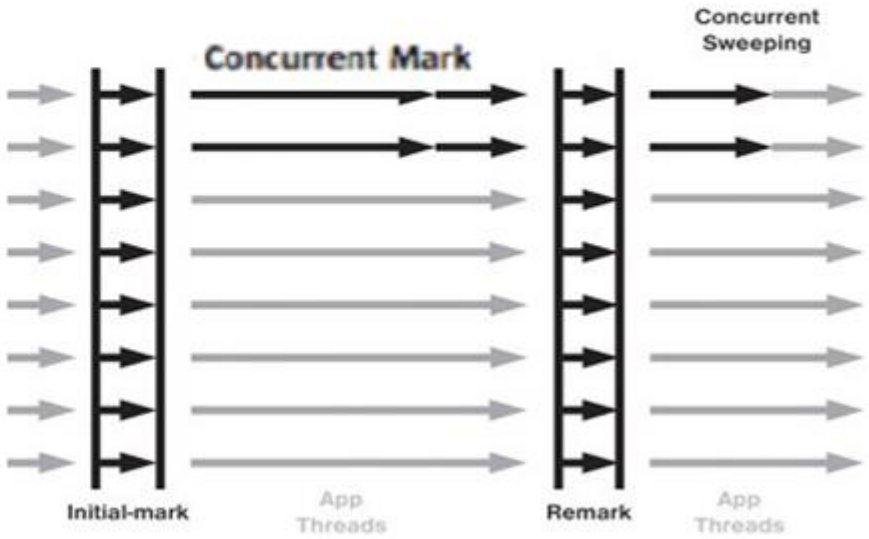
A recommended setting for JVM looks like following

```
-Xmx8g -Xms8g -XX:MetaspaceSize=96m
-XX:+UseG1GC -XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35
-XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80
```

* Ref to kafka-run-class for these settings

Parallel GC

Concurrent Mark-Sweep Collector



-XX:+PrintCommandLineFlags

Disks And File System

It is recommend to use multiple drives to get good throughput. Do not share the same drives with any other application or for kafka application logs.

Multiple drives can be configured using log.dirs in server.properties. Kafka assigns partitions in round-robin fashion to log.dirs directories.

Zookeeper

- ✓ Do not co-locate zookeeper on the same boxes as Kafka
- ✓ It is recommend zookeeper to isolate and only use for Kafka not any other systems should be depend on this zookeeper cluster
- ✓ Make sure you allocate sufficient JVM , good starting point is 4Gb
- ✓ Monitor: Use JMX metrics to monitor the zookeeper instance

Choosing Topic/Partitions

- ✓ Topic/Partition is unit of parallelism in Kafka
- ✓ Partitions in Kafka drives the parallelism of consumers
- ✓ Higher the number of partitions more parallel consumers can be added , thus resulting in a higher throughput.
- ✓ Based on throughput requirements one can pick a rough number of partitions.

    - Lets call the throughput from producer to a single partition is P
    - Throughput from a single partition to a consumer is C
    - Target throughput is T
    - Required partitions = Max (T/P, T/C)

More partitions can increase the latency

The end-to-end latency in Kafka is defined by the time from when a message is published by the producer to when the message is read by the consumer.

Kafka only exposes a message to a consumer after it has been committed, i.e., when the message is replicated to all the in-sync replicas.

Replication 1000 partitions from one broker to another can take up 20ms. This can be too high for some real-time applications

ACKs

Defines durability level for producer.

Acks	Throughput	Latency	Durability
0	High	Low	No Gurantee
1	Medium	Medium	Leader
-1	Low	High	ISR

Partitions and Memory Usage

- ✓ Brokers allocate a buffer the size of **replica.fetch.max.bytes** for each partition they replicate.
- ✓ If replica.fetch.max.bytes is set to 1 MB, and you have 1000 partitions, about 1 GB of RAM is required.
- ✓ Ensure that the number of partitions multiplied by the size of the largest message does not exceed available memory.

- ✓ The same consideration applies for the consumer **fetch.message.max.bytes** setting. Ensure that you have enough memory for the largest message for each partition the consumer replicates.

- ✓ With larger messages, you might need to use fewer partitions or provide more RAM.

- ✓ **Note : Kafka brokers** use both the JVM heap and the OS page cache. The JVM heap is used for replication of partitions between brokers and for log compaction.

Partition Reassignment

If we add a new Kafka broker to the existing cluster to handle increased demand, new partitions are allocated to it, but it does not automatically share the load of existing partitions on other brokers.

To redistribute the existing load among brokers, we must manually reassign partitions. We can do so using **kafka-reassign-partitions.sh** script utilities.

To reassign partitions:

1. Create a list of topics you want to move.

topics-to-move.json
```
{"topics": [{"topic": "foo1"},
                {"topic": "foo2"}],
 "version":1
}
```

2. Use the --generate option in **kafka-reassign-partitions.sh** to list the distribution of partitions and replicas on your current brokers, followed by a list of suggested locations for partitions on your new broker.

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181
    --topics-to-move-json-file topics-to-move.json
    --broker-list "4"
    --generate

Current partition replica assignment

{"version":1,
 "partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
               {"topic":"foo1","partition":0,"replicas":[3,1]},
               {"topic":"foo2","partition":2,"replicas":[1,2]},
               {"topic":"foo2","partition":0,"replicas":[3,2]},
               {"topic":"foo1","partition":1,"replicas":[2,3]},
               {"topic":"foo2","partition":1,"replicas":[2,3]}]
}

{"version":1,
 "partitions":[{"topic":"foo1","partition":3,"replicas":[4]},
               {"topic":"foo1","partition":1,"replicas":[4]},
               {"topic":"foo2","partition":2,"replicas":[4]}]
}
```

3. Revise the suggested list if required, and then save it as a JSON file.

4. Use the --execute option in kafka-reassign-partitions.sh to start the redistirbution process, which can take several hours in some cases.

```
> bin/kafka-reassign-partitions.sh \
    --zookeeper localhost:2181 \
    --reassignment-json-file expand-cluster-reassignment.json
    --execute
```

Use the --verify option in kafka-reassign-partitions.sh to check the status of your partitions.

Although reassigning partitions is labor-intensive, we should anticipate system growth and redistribute the load when your system is at 70% capacity.

If we wait until we are forced to redistribute because we have reached the limit of your resources, the redistribution process can be extremely slow.

Handling Large Messages

Before configuring Kafka to handle large messages, first consider the following options to reduce message size:

- ✓ The Kafka producer can compress messages. For example, if the original message is a text-based format (such as XML), in most cases the compressed message will be sufficiently small.

- ✓ Use the offsets.topic.compression.codec and compressed.type producer configuration parameters to enable compression. Gzip and Snappy are supported.

- ✓ If shared storage (such as NAS, HDFS, or S3) is available, consider placing large files on the shared storage and using Kafka to send a message with the file location. In many cases, this can be much faster than using Kafka to send the large file itself.

- ✓ Split large messages into 1 KB segments with the producing client, using partition keys to ensure that all segments are sent to the same Kafka partition in the correct order. The consuming client can then reconstruct the original large message.

If we still need to send large messages with Kafka, modify the configuration parameters presented in the following sections to match the requirements.

Broker Configuration Properties

Property	Default Value	Description
`message.max.bytes`	1000000 (1 MB)	Maximum message size the broker accepts. When using the old consumer, this property must be lower than the consumer `fetch.message.max.bytes`, or the consumer cannot consume the message.
`log.segment.bytes`	1073741824 (1 GiB)	Size of a Kafka data file. Must be larger than any single message.
`replica.fetch.max.bytes`	1048576 (1 MiB)	Maximum message size a broker can replicate. Must be larger than `message.max.bytes`, or a broker can accept messages it cannot replicate, potentially resulting in data loss.

Consumer Configuration

Kafka offers two separate consumer implementations, the old consumer and the new consumer.

The old consumer is the Consumer class written in Scala.

The new consumer is the KafkaConsumer class written in Java.

When configuring Kafka to handle large messages, different properties have to be configured for each consumer implementation.

Old Consumer Configuration Properties

Property	Default Value	Description
`fetch.message.max.bytes`	52428800 (50 MiB)	The maximum amount of data the server should return for a fetch request. This is a hard limit. If a message batch is larger than this limit, the consumer will not be able to consume the message or any subsequent messages in a given partition.

New Consumer Configuration Properties

Property	Default Value	Description
`max.partition.fetch.bytes`	1048576 (10 MiB)	The maximum amount of data per-partition the server will return.
`fetch.max.bytes`	52428800 (50 MiB)	The maximum amount of data the server should return for a fetch request.

Note: The new consumer is able to consume a message batch that is larger than the default value of the max.partition.fetch.bytes or fetch.max.bytes property. However, the batch will be sent alone, which can cause performance degradation.

Tuning Kafka for Optimal Performance

Performance tuning involves two important metrics:

Latency measures how long it takes to process one event, and throughput measures how many events arrive within a specific amount of time.

Most systems are optimized for either latency or throughput. Kafka is balanced for both.

A well tuned Kafka system has just enough brokers to handle topic throughput, given the latency required to process information as it is received.

Tuning the producers, brokers, and consumers to send, process, and receive the largest possible batches within a manageable amount of time results in the best balance of latency and throughput for the Kafka cluster.

Tuning Kafka Producers

Suppose the network roundtrip time between our application and the Kafka cluster is 10ms.

If we wait for a reply after sending each message, sending 100 messages will take around 1 second. (Synchronous)

On the other hand, if we just send all our messages and not wait for any replies, then sending 100 messages will barely take any time at all. (Fire-and-Forget)

On the other hand, we do need to know when we failed to send a message completely so we can throw an exception, log an error, or perhaps write the message to an “errors” file for later analysis. In order to send messages asynchronously and still handle error scenarios, the producer supports adding a callback when sending a record. (Asynchronous)

Batch Size

batch.size measures batch size in total bytes instead of the number of messages. It controls how many bytes of data to collect before sending messages to the Kafka broker. Set this as high as possible, without exceeding available memory. The default value is 16384.

If you increase the size of your buffer, it might never get full. The Producer sends the information eventually, based on other triggers, such as linger time in milliseconds. Although you can impair memory usage by setting the buffer batch size too high, this does not impact latency.

If your producer is sending all the time, you are probably getting the best throughput possible. If the producer is often idle, you might not be writing enough data to warrant the current allocation of resources.

Linger Time

linger.ms sets the maximum time to buffer data in asynchronous mode. For example, a setting of 100 batches 100ms of messages to send at once. This improves throughput, but the buffering adds message delivery latency.

By default, the producer does not wait. It sends the buffer any time data is available.

Instead of sending immediately, you can set linger.ms to 5 and send more messages in one batch. This would reduce the number of requests sent, but would add up to 5 milliseconds of latency to records sent, even if the load on the system does not warrant the delay.

Increase linger.ms for higher latency and higher throughput in your producer.

Tuning Kafka Consumers

Consumers can create throughput issues on the other side of the pipeline. The maximum number of consumers for a topic is equal to the number of partitions. We need enough partitions to handle all the consumers needed to keep up with the producers.

Consumers in the same consumer group split the partitions among them. Adding more consumers to a group can enhance performance. Adding more consumer groups does not affect performance.

Kafka Consumer Load Share

Kafka consumer consumption divides partitions over consumer instances within a consumer group.

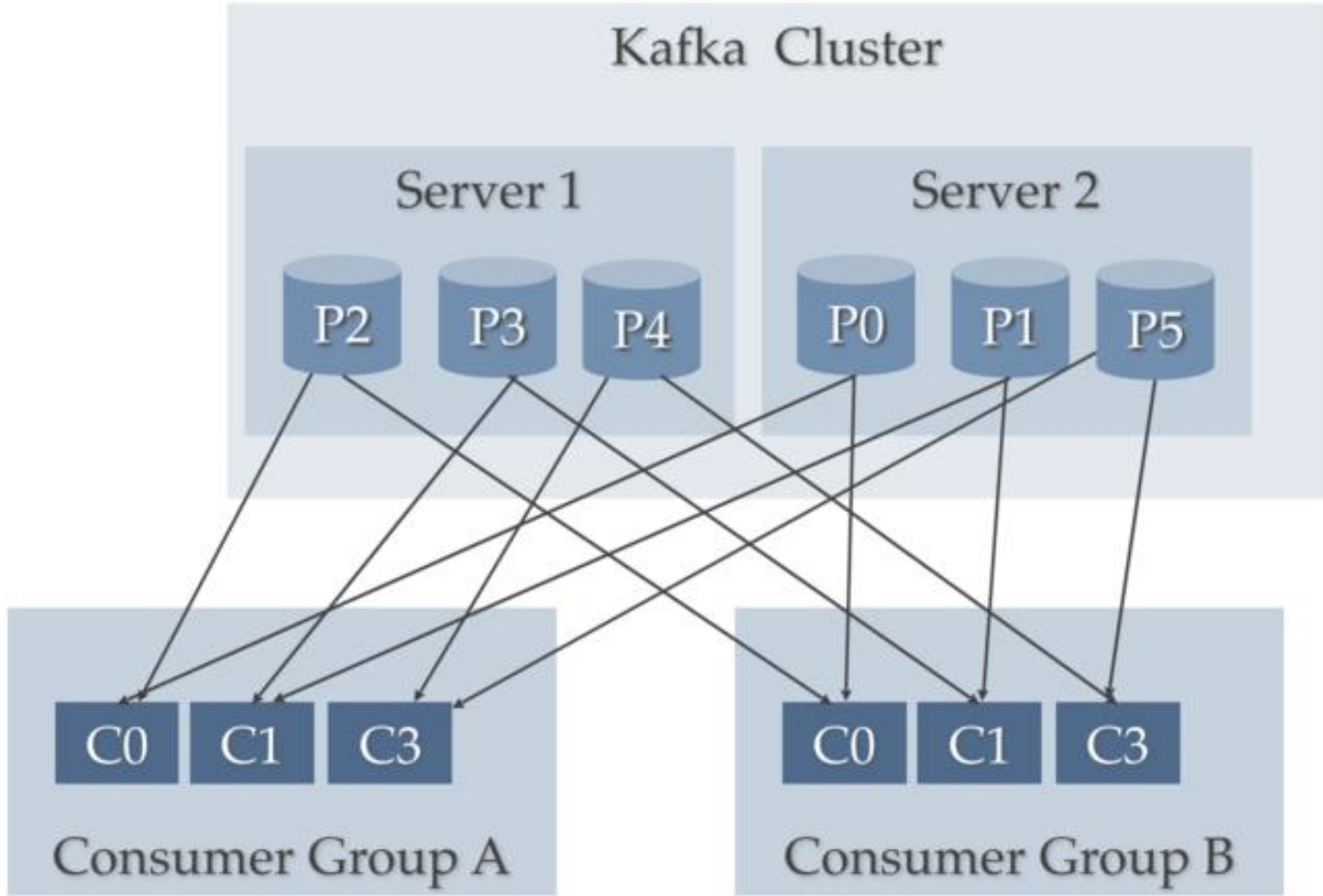
Each consumer in the consumer group is an exclusive consumer of a “fair share” of partitions. This is how Kafka does load balancing of consumers in a consumer group.

Consumer membership within a consumer group is handled by the Kafka protocol dynamically.

If new consumers join 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 Cluster
Server 1
Server 2
P2
P3
P4
P0
P1
P5
C0
C1
C3
C0
C1
C3
Consumer Group A
Consumer Group B

Thread per Consumer

If you need to run multiple consumers, then run each consumer in their own thread.

This way Kafka can deliver record batches to the consumer and the consumer does not have to worry about the offset ordering.

A thread per consumer makes it easier to manage offsets. It is also simpler to manage failover (each process runs X num of consumer threads) as you can allow Kafka to do the brunt of the work.

Kafka Consumer Failover

Consumers notify the Kafka broker when they have successfully processed a record, which advances the offset.

If a consumer fails before sending commit offset to Kafka broker, then a different consumer can continue from the last committed offset.

If a consumer fails after processing the record but before sending the commit to the broker, then some Kafka records could be reprocessed. In this scenario, Kafka implements the at least once behaviour, and you should make sure the messages (record deliveries) are idempotent

Consumption Performance

The main way we scale data consumption from a Kafka topic is by adding more consumers to a consumer group. It is common for Kafka consumers to do high-latency operations such as write to a database or a time-consuming computation on the data.

This is a good reason to create topics with a large number of partitions—it allows adding more consumers when the load increases.

Kafka At LinkedIn

- 1100+ Kafka brokers
- Over 32,000 topics
- 350,000+ Partitions

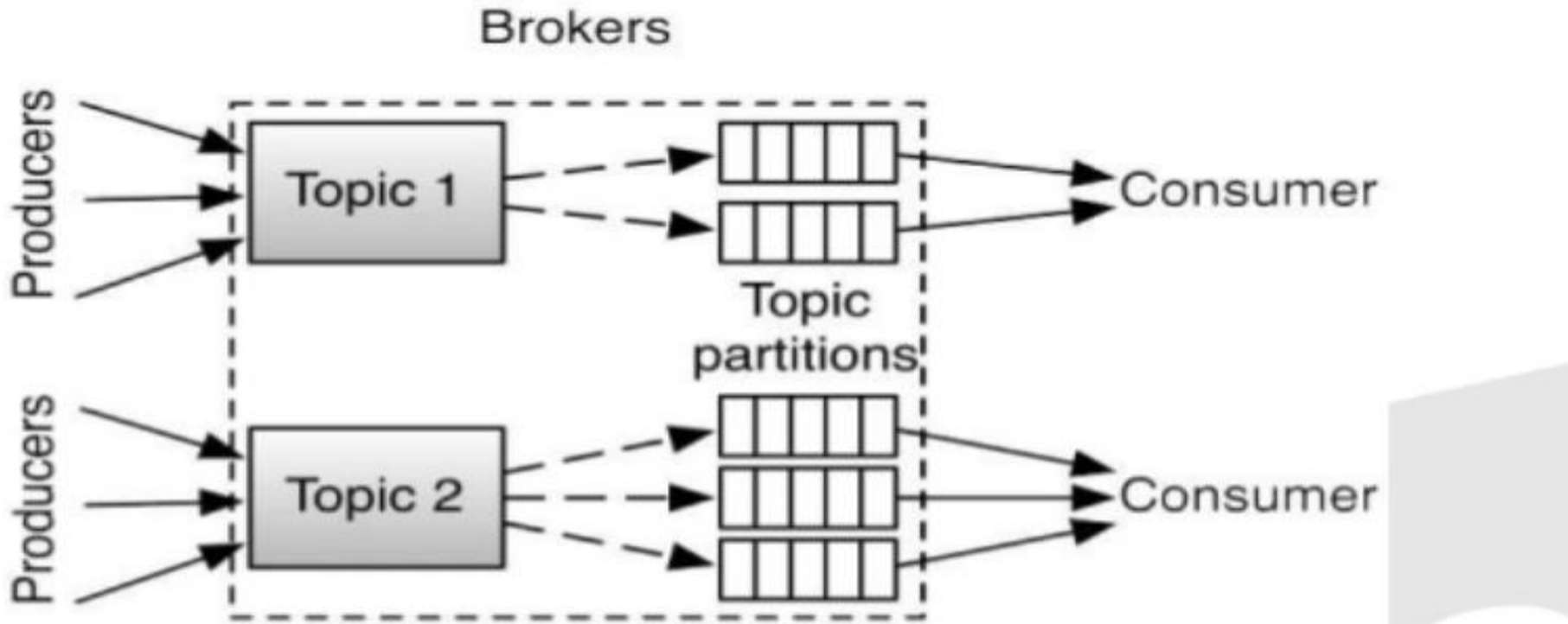
- 875 Billion messages per day
- 185 Terabytes In
- 675 Terabytes Out

- Peak Load
  - 10.5 Million messages/sec
  - 18.5 Gigabits/sec Inbound
  - 70.5 Gigabits/sec Outbound

Key Idea #1:

Data-parallelism leads to scale-out

Distribute Clients across Partitions

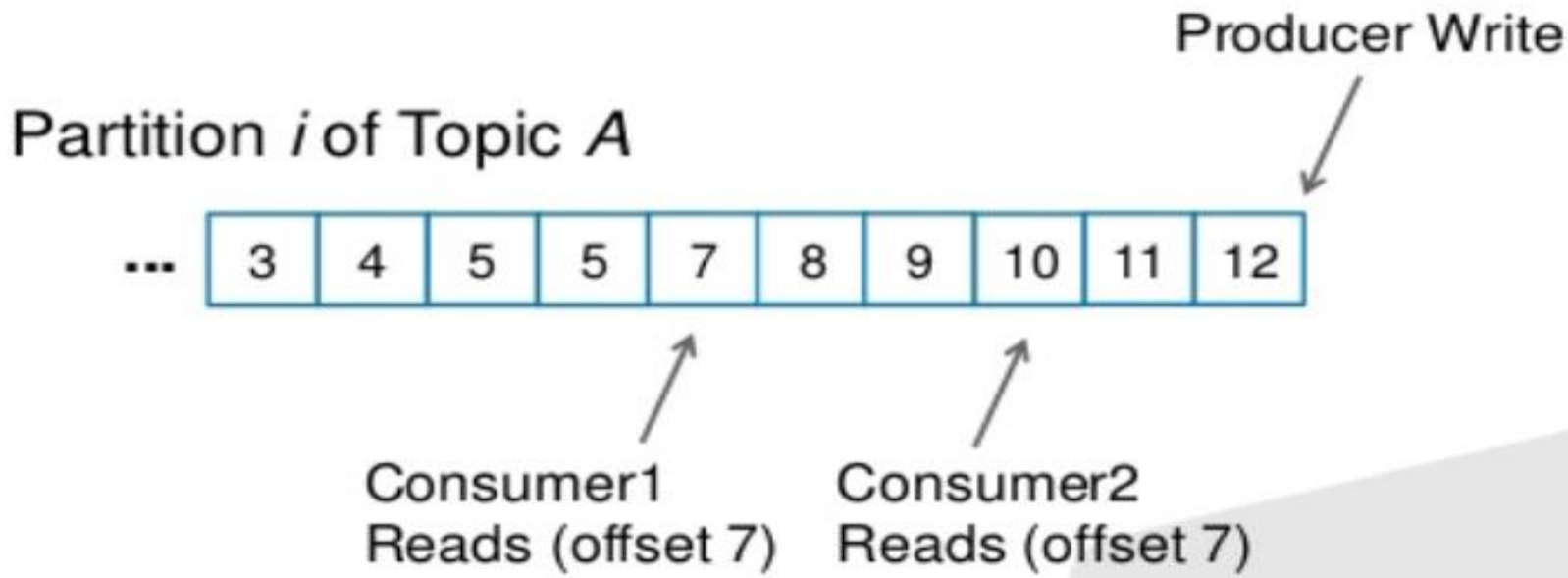


- Produce/consume requests are randomly balanced among brokers

Key Idea #2:

Disks are fast when used sequentially

Store Messages as a Log



- Appends are effectively O(1)
- Reads from known offset are fast still, when cached

Key Idea #3:

Batching makes best use of network/IO

Batch Transfer

- Batched send and receive
- Batched compression
- No message caching in JVM
- Zero-copy from file to socket (Java NIO)

Out-of-Sync Replicas

Seeing one or more replicas rapidly flip between in-sync and out-of-sync status is a sure sign that something is wrong with the cluster.

The cause is often a misconfiguration of Java's garbage collection on a broker. Misconfigured garbage collection can cause the broker to pause for a few seconds, during which it will lose connectivity to Zookeeper.

When a broker loses connectivity to Zookeeper, it is considered out-of-sync with the cluster.

Replication Factor (default is 1)

The topic-level configuration is replication.factor. At the broker level, you control the default.replication.factor for automatically created topics.

A reasonable assumption, topics had a replication factor of three, meaning that each partition is replicated three times on three different brokers

A replication factor of N allows you to lose N-1 brokers while still being able to read and write data to the topic reliably. So a higher replication factor leads to higher availability, higher reliability, and fewer disasters.

On the flip side, for a replication factor of N, you will need at least N brokers and you will store N copies of the data, meaning we will need N times as much disk space.

How do you determine the right number of replicas for a topic?

The answer is based on how critical a topic is and how much you are willing to pay for higher availability.

Placement of Replicas

By default, Kafka will make sure each replica for a partition is on a separate broker.

However, in some cases, this is not safe enough. If all replicas for a partition are placed on brokers that are on the same rack and the top-of-rack switch misbehaves, you will lose availability of the partition regardless of the replication factor.

To protect against rack-level misfortune, it is recommend placing brokers in multiple racks and using the broker.rack broker configuration parameter to configure the rack name for each broker.

broker.rack=my-rack-id

Unclean Leader Election

This configuration is only available at the broker (and in practice, cluster-wide) level. The parameter name is unclean.leader.election.enable and by default it is set to false.

When the leader for a partition is no longer available, one of the in-sync replicas will be chosen as the new leader. This leader election is “clean” in the sense that it guarantees no loss of committed data

when no in-sync replica exists except for the leader that just became unavailable?

This situation can happen in one of two scenarios:

1). The partition had three replicas, and the two followers became unavailable, In this situation, as producers continue writing to the leader, all the messages are acknowledged and committed.

Suppose, that the leader becomes unavailable??
In this scenario, if one of the out-of-sync followers starts first, we have an out-of-sync replica as the only available replica for the partition.

2). The partition had three replicas and, due to network issues, the two followers fell behind so that even though they are up and replicating, they are no longer in sync. The leader keeps accepting messages as the only in-sync replica.

Now if the leader becomes unavailable, the two available replicas are no longer in-sync.

In both these scenarios, we need to make a difficult decision:

1). If we don’t allow the out-of-sync replica to become the new leader, the partition will remain offline until we bring the old leader (and the last in-sync replica) back online.

In some cases (e.g., memory chip needs replacement), this can take many hours.

2).If we do allow the out-of-sync replica to become the new leader, we are going to lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers.

Imagine that while replicas 0 and 1 were not available, we wrote messages with offsets 100-200 to replica 2 (then the leader).

Now replica 2 is unavailable and replica 0 is back online.

Replica 0 only has messages 0-100 but not 100-200. If we allow replica 0 to become the new leader, it will allow producers to write new messages and allow consumers to read them.

So, now the new leader has completely new messages 100-200.

First, let’s note that some consumers may have read the old messages 100-200, some consumers got the new 100-200, and some got a mix of both.

This can lead to pretty bad consequences when looking at things like downstream reports.

Setting **unclean.leader.election.enable** to true means we allow out-of-sync replicas to become leaders (knowns as unclean election), knowing that we will lose messages when this occurs.

If we set it to false, we choose to wait for the original leader to come back online, resulting in lower availability

Minimum In-Sync Replicas

Both the topic and the broker-level configuration are called min.insync.replicas.

If you would like to be sure that committed data is written to more than one replica, We need to set the minimum number of in-sync replicas to a higher value.

If a topic has three replicas and you set min.insync.replicas to 2, then you can only write to a partition in the topic if at least two out of the three replicas are in-sync.

When all three replicas are in-sync, everything proceeds normally. This is also true if one of the replicas becomes unavailable.

However, if two out of three replicas are not available, the brokers will no longer accept produce requests. Instead, producers that attempt to send data will receive NotEnoughReplicasException.

Consumers can continue reading existing data. In effect, with this configuration, a single in-sync replica becomes read-only.

This prevents the undesirable situation where data is produced and consumed, only to disappear when unclean election occurs.

In order to recover from this read-only situation, we must make one of the two unavailable partitions available again (maybe restart the broker) and wait for it to catch up and get in-sync.

Enforcing Client Quotas
[Flow Controller]

Kafka cluster has the ability to enforce quotas on produce and fetch requests.

Quotas are basically byte-rate thresholds defined per client-id.

A client-id logically identifies an application making a request.

Hence a single client-id can span multiple producer and consumer instances and the quota will apply for all of them as a single entity i.e. if client-id=”test-client” has a produce quota of 10MB/sec, this is shared across all instances with that same id.

Quotas protect brokers from producers and consumers who produce/consume very high volumes of data and thus monopolize broker resources and cause network saturation.

This is especially important in large multi-tenant clusters where a small set of badly behaved clients can degrade user experience for the well behaved ones.

In fact, when running Kafka as a service quotas make it possible to enforce API limits according to an agreed upon contract.

By default, each unique client-id receives a fixed quota in bytes/sec as configured by the cluster (quota.producer.default, quota.consumer.default).

This quota is defined on a per-broker basis. Each client can publish/fetch a maximum of X bytes/sec per broker before it gets throttled.

When a broker detects quota violation, it does not return an error. Rather it attempts to slow down a client exceeding its quota.

The broker computes the amount of delay needed to bring the quota-violating client under it’s quota and delays the response for that time.

This approach keeps the quota violation transparent to clients

By default, clients receive an unlimited quota. It is possible to set custom quotas for each (user, client-id), user or client-id group.

1). Configure custom quota for client-id=clientA:

- kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048, request_percentage=200' --entity-type clients --entity-name clientA

Updated config for entity: client-id 'clientA'.

2). Configure custom quota for user=user1:

- kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'producer_byte_rate=1024,consumer_byte_rate=2048, request_percentage=200' --entity-type users --entity-name user1

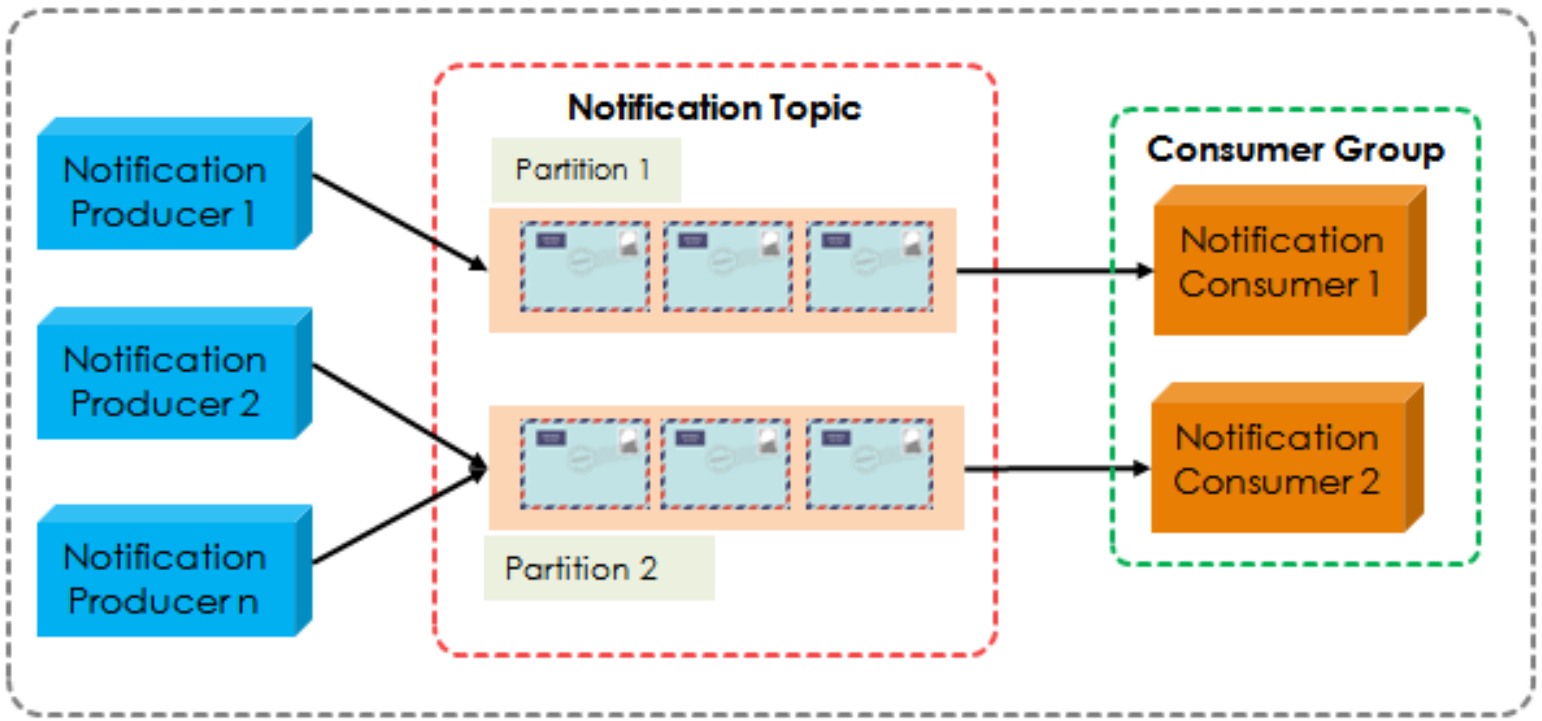
producer_byte_rate: This quota limits the number of bytes that a producer application is allowed to send per second.

consumer_byte_rate: This quota limits the number of bytes that a consumer application is allowed to receive per second.

request_percentage: The percentage per quota window (out of a total of (num.io.threads + num.network.threads) * 100%) for requests from the user or client, above which the request may be throttled.

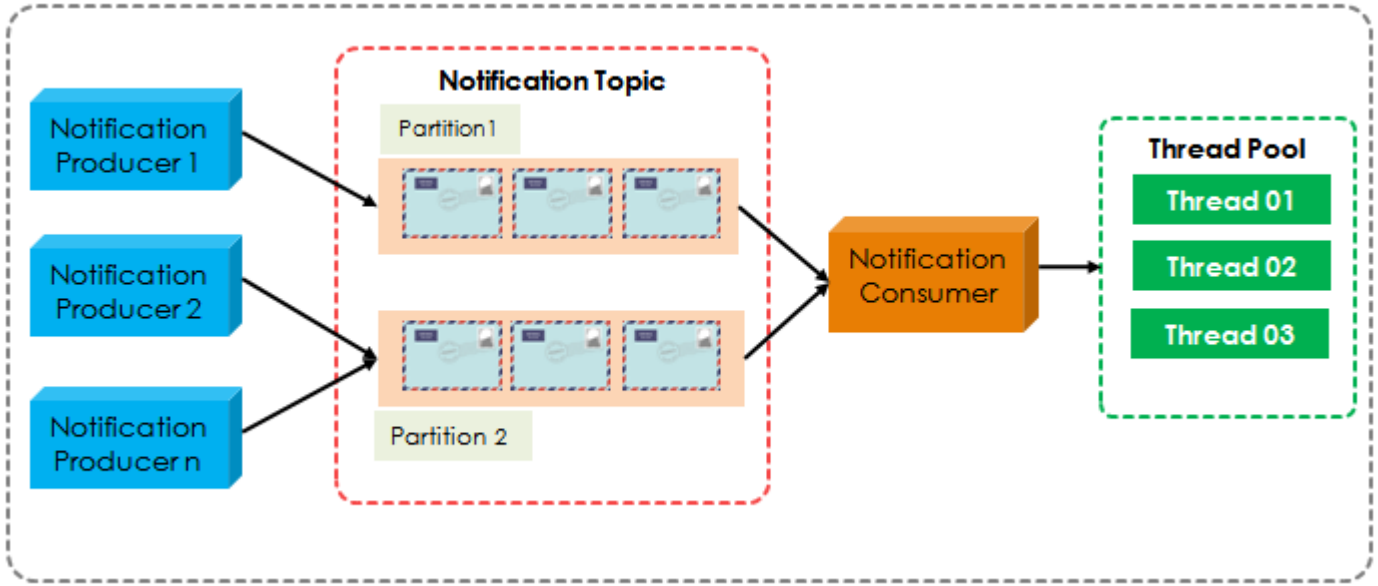
num.io.threads -> default 8
num.network.threads -> 3

Multiple consumers with their own threads



Pros	Cons
Easy to implement	The total of consumers is limited the total partitions of the topic.
Implementing in-order processing on per-partition is easier.	More TCP connections to the brokers

Single consumer, multiple worker processing threads



Pros	Cons
Be flexible in scale out the number of processing thread	It’s not easy to implement in-order processing on per partition. Let’s say there are 2 messages on the same partitions being processed by 2 different threads. To guarantee the order, those 2 threads must be coordinated somehow.