System Design: Zookeeper + Kafka:

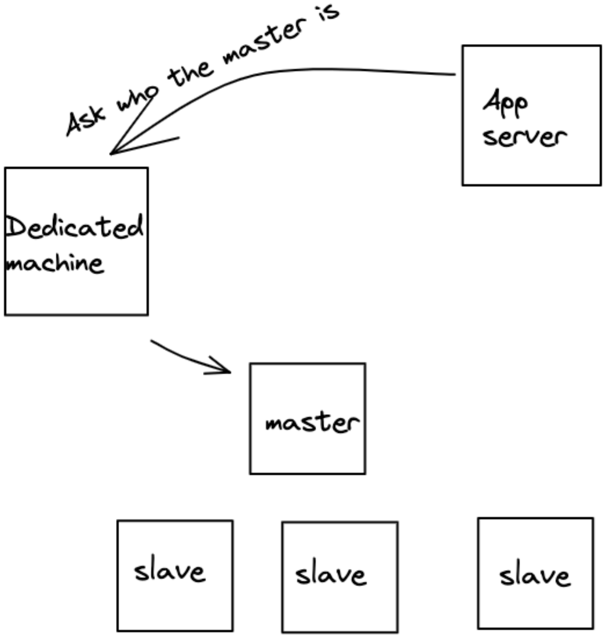
Problem Statement 1 (State tracking):

In a Master Slave architecture, all writes must come to the master and not the slave machines. This means all clients (appservers) must be aware of who the master is. As long as the master is the same, that's not an issue.

The problem is, If the master might die, and in that case we want to select a new master and all the machines should be aware of it, they should be in sync.

If you were to think of this as a problem statement, how would you solve it?

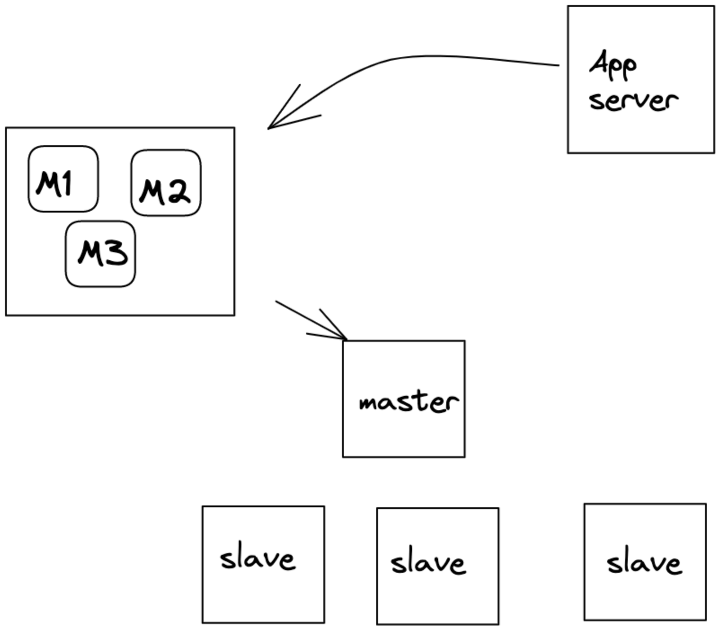
A naive approach might be to say that we will have a Dedicated machine and the only job of this machine is to keep track of who the master is. Anytime an appserver wants to know who the master is, they go and ask this dedicated machine.



However, there are 2 issues with this approach.

1. This dedicated machine will become the single point of failure. If the machine is down, no writes can happen - even though the master might be healthy.
2. For every request, we have introduced an additional hop to find out who the master is.

To solve the issue #1, maybe instead of 1 machine we can use a bunch of machines or clusters of machines.

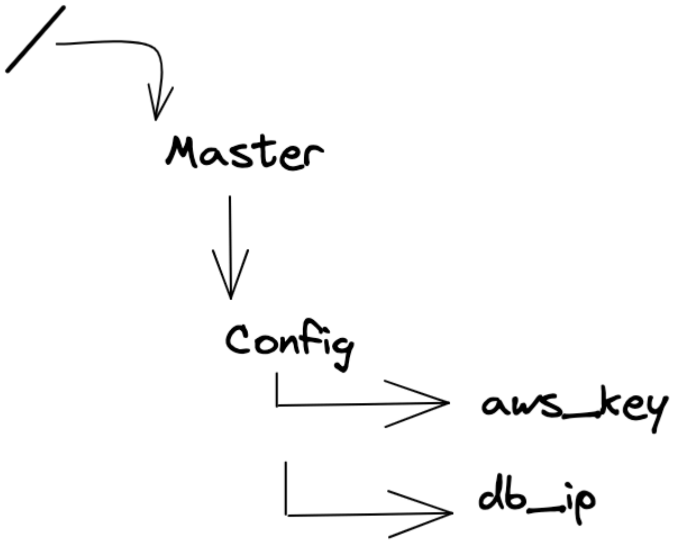


- *How do these machines find out who the master is?*
- *How do we make sure that all these machines have the same information about the master?*
- *How do we enable appservers to directly go to master without the additional hop to these machines?*

Solution: Zookeeper:

Zookeeper is a generic system that tracks data in **strongly consistent** form. More on this later.

Storage in Zookeeper is exactly like a file system.
Example, we have a root folder inside that we have bunch of files or directories.



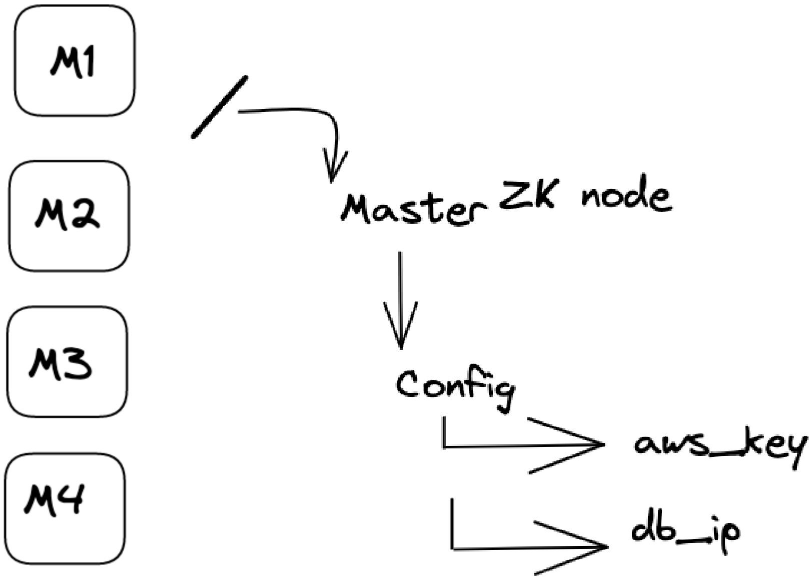
All these files are known as ZK nodes in zookeeper.

ZK Nodes:
Every file in zookeeper is of one of two kinds:

1. Ephemeral : Ephemeral nodes ***(do not confuse node to mean a machine. Nodes are files in the context of Zookeeper)*** are files where the data written is only valid till the machine/session which wrote the data is alive/active. This is a fancier way of saying that the machine which wrote on this node has to keep sending heartbeat to ensure the data on this node is not deleted.
    a. Once an ephemeral node is written, other machines / sessions cannot write any data on it. An ephemeral node has exactly one session/machine as the owner. Only the owner can modify the data.
    b. When the owner does not send a heartbeat, the session dies and the ephemeral node is deleted. This means any other machine can then create the same node/file with different data.
    c. These are the nodes which are used to track machine status, master of a cluster, taking a distributed lock, etc. More on this later.

2. Persistent : Persistent nodes are nodes where the node is not deleted unless specifically requested to be deleted. These nodes are used to store configuration variables.

ZK Node for consistency / Master Election:
To keep things simple, let's imagine that Zookeeper is a single machine (we will move to multiple machines later). Let's imagine there are a bunch of storage machines in a cluster X.



They all want to become master. However, there can only be one master. So, how do we resolve the “kaun banega master” challenge. We ask all of them to try to write their IP address as data to the same ephemeral ZK node (let’s say /clusterx/master_ip).

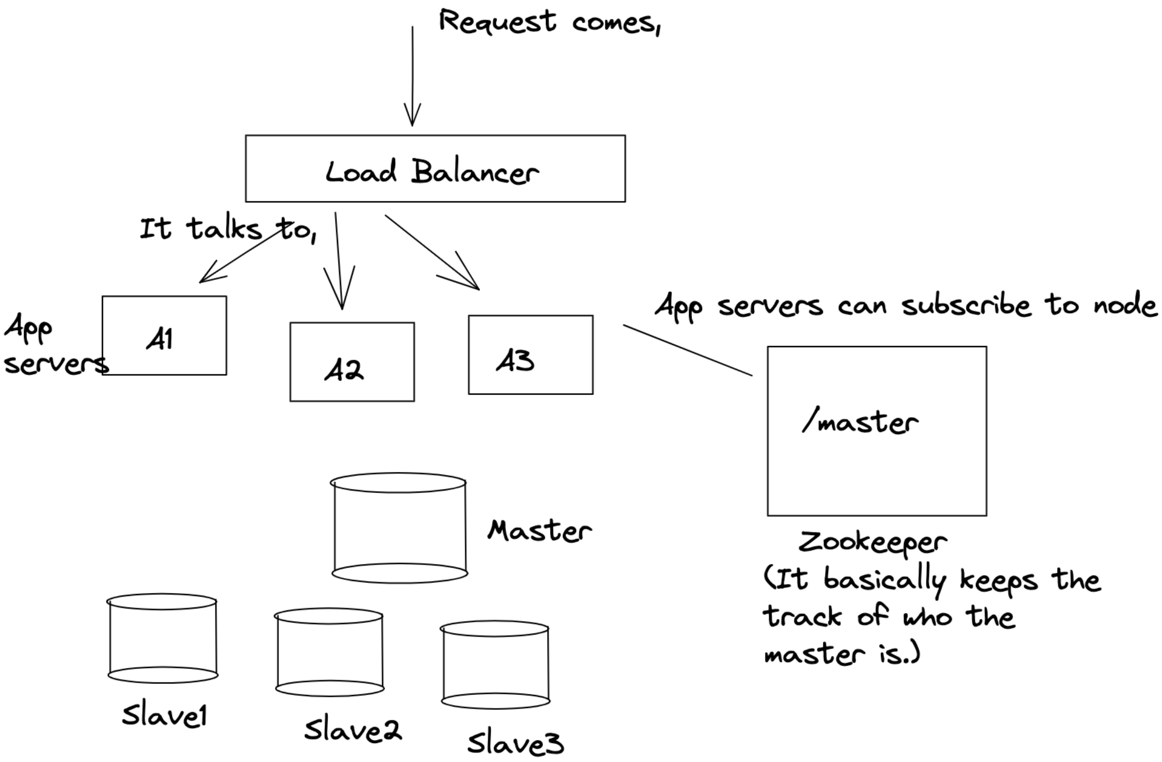
Note that only one node will be able to write to this ephemeral node and all other writes will fail.
So, let’s say M2 was able to write M2’s ip address on /clusterx/master_ip.
Now, as long as M2 is alive and keeps sending heartbeat, /clusterx/master_ip will have M2’s ip address. When any machine tries to read data on /clusterx/master_ip, they will get M2’s ip address in return.

ZK: Setting a watch:
There is still the additional hop problem. If all appservers and other machines have to talk to the zookeeper on every request to find out who the master is, not only does it add so much load to zookeeper, it also increases hops of every request.

How can we address that?
If you think about it, the data on ephemeral node changes very less frequently (probably like once in a day - not even that). It seems stupid that every client has to come to ZK to ask for the master value when it does not change most of the time.
So, how about we reverse the process. We tell people, “Here is the value X. No need to keep asking me again and again. Keep using this value. Whenever this value gets updated, I will notify you.”.

Zookeeper does a similar thing. It solves that using a “subscribe to updates on this ZK node” option.
On any ZK node, you can **set a watch** (subscribe to updates). In ZooKeeper, all of the read operations have the option of setting a watch as a side effect.
If I am an appserver, and I set a watch on /clusterx/master_ip, then when this node data changes or this node gets deleted, I (and all other clients who had set a watch on that node) will be notified. This means when clients set a watch, zookeeper maintains a list of subscribers (per node/file).

ZK: Architecture:
All of this is great. But we were assuming ZK is a single machine. But ZK cannot be a single machine. How does this work across multiple machines?



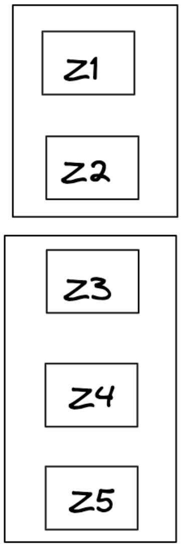
Now the problem is Zookeeper is a single machine and if it’s a single machine it becomes a single point of failure.
Hence zookeeper is actually a bunch of machines **(odd number of machines**).

Zookeeper machines also select a leader/master among themselves. When you setup the set of machines (or when the existing leader dies in case of running cluster), the first step is electing the leader. [How is a leader elected in Apache ZooKeeper? - Quora / ZK Leader Election Code for the curious ones to explore how leader election happens in Zookeeper].

Now, let's say Z3 is elected as leader, whenever any write is done, for Eg, change /master ip address to ip,x then it is first written to the leader and leader broadcasts that change to all other machines. If at least the majority of the machines (including the leader) acknowledge the change, then the write is considered successful, otherwise it is rolled back.
So, in a cluster of 5 machines, 3 machines need to acknowledge for the write to succeed (in a cluster of 7, 4 machines need to acknowledge and so forth). *Note that even if a machine dies, the total number of machines still stays 5, and hence even then 3 machines need to acknowledge.*

Hence, if 10 machines were trying to become the master and they all sent requests to write to /clusterx/master simultaneously, all those requests would come to a single machine - ***the leader,*** first. The leader can implement a lock to ensure only one of those requests goes through first, and the data is written if majority ZK machines acknowledge. Else data is rolled back, lock released and then the next request gets the lock.

But why the majority number of machines?
Let's imagine we let the write succeed if it succeeds on X/2 number of machines (X being the total number of machines). For this let’s imagine we have 5 zookeeper machines, and because of network partition z1 and z2 become disconnected from the other 3 machines.



Let’s say write1 (/clusterx/master_ip = ip1) happens on z1 and z2. .
Let’s say another write write2 (/clusterx/master_ip = ip2) happens for the same ZK node on z4 and z5.

Now when we try to read (/clusterx/master_ip) then half of the machines would suggest ip1 is master, and the other half would return ip2 as master. This is called **split brain.**
Hence we need Quorum / Majority so that we don’t end up having 2 sets of machines saying x is the answer or y is the answer, there should be consistency.
So till the write is not successful on majority of the machines we can’t return success, now in this case both ip1 and ip2 try to write in Z3 and whoever succeeds the master will have that address and the other one fails.

ZK: Master dies
Imagine master had written it’s ip address to */clusterx/master_ip*. All appservers and slaves had set watch on the same node (noting down the current master IP address).

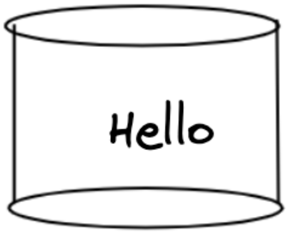
Imagine the master dies. What happens?

- Master machine won’t be able to send heartbeat to the zookeeper for the ephemeral node */clusterx/master_ip*
- The ephemeral node */clusterx/master_ip* will hence be deleted.
- All subscribers will be notified of the change.

- Slaves, as soon as they get this update, will try to become masters again. Whoever is the first one to write on Zookeeper becomes the new master.
- Appserver will delete the local value of master_ip. They would have to read from the zookeeper (+set new watch + update local master_ip value) whenever the new write request comes.
    - If they get back null as value, the request fails. New master is not selected yet.
- Old master whenever it comes back up will read from the same ZK node to find out the new master machine and will become a slave itself.
    - Unless it comes back up quickly, finds ZK node to be null and tries along with other slaves to become the new master.

Problem Statement 2 (Async tasks):

Let's take an example of the messenger,
Imagine whenever a message comes for a user, say abhi sends a message to raj, so the message is written in the raj database.



After this we want to do a couple of things.

1. Notify raj
2. Email to raj
   (If raj is not reading messages for the last 24 hrs).
3. Update relevant metrics in analytics

Now whenever a message comes, we have to do these things but we don't want that the sender of the message to wait for these things to happen. Infact, if any of the above fails, it does not mean that the message sent itself failed.
So how to return success immediately?

To solve these types of problems where we have to do a few things asynchronously we use something known as **Persistent Queue.**
Persistent Queue is durable which means we are actually writing it in a hard disk so that we won't lose it.

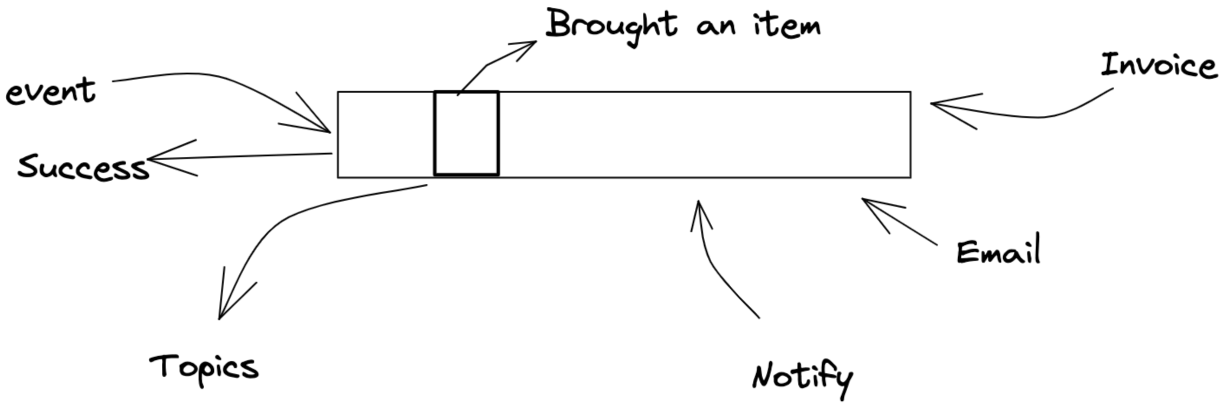
Solution: Persistent Queue:
Persistent Queues work on a model called pub-sub (Publish Subscribe).

PubSub:
Pubsub has 2 parts:

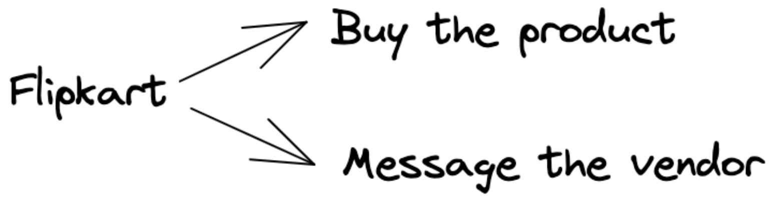
- **Publish:** You look at all events of interest that would require actions post it. For example, a message being sent is an event. Or imagine someone buys an item on Flipkart. That could be an event. You publish that event on a persistent queue.
- **Subscriber:** Different events could have different kind of subscribers interested in that event. They ***consume*** events they have subscribed to from the queue. For example, in the above example, message notification system, message email system and message analytics system would subscribe to the event of “a message sent” on the queue.
    - Or an invoice generation system could subscribe to the event of “bought an item on Flipkart”.

There could be multiple types of events being published, and each event could have multiple kind of subscriber consuming these events.

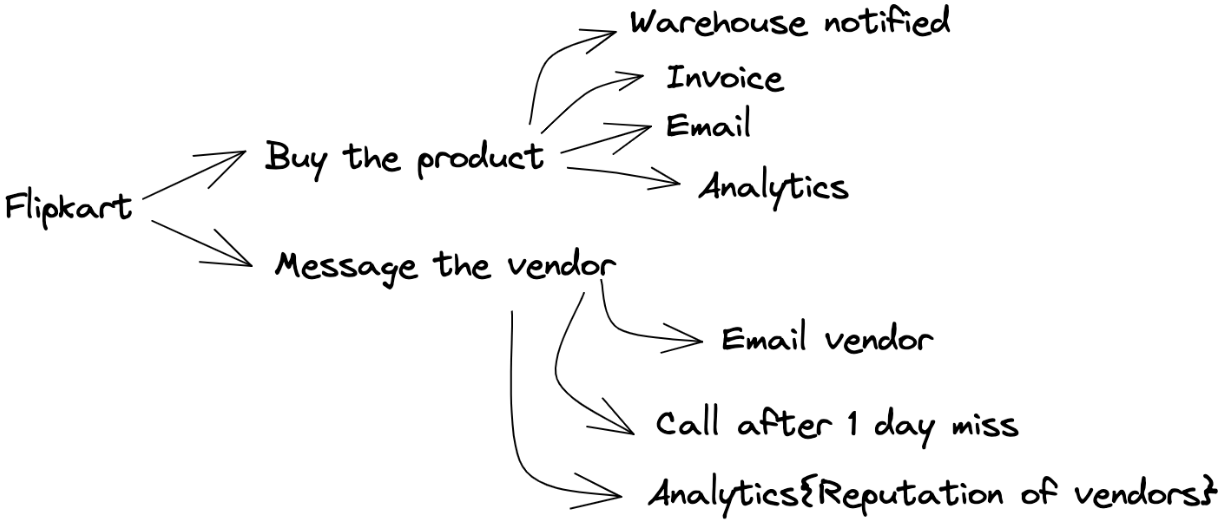
Topics:
Now within a queue also we need some segregation of topics because the system doesn't want to subscribe to the whole queue, they need to subscribe to some particular type of event and each of these events is called topic.



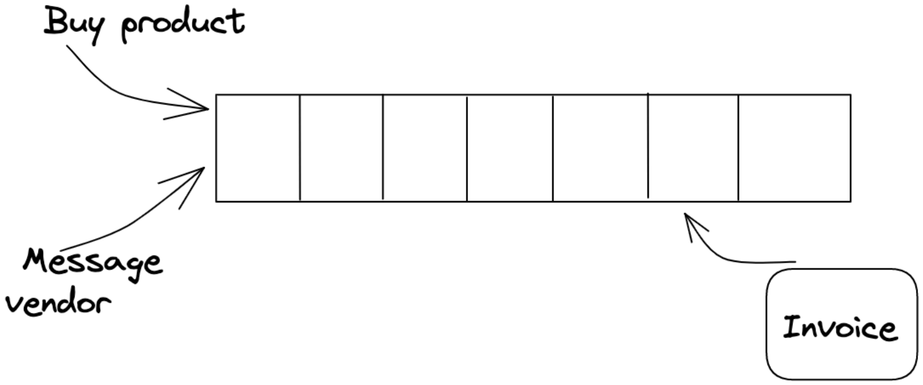
Let’s take an example of Flipkart,
Say, flipkart also has an inbuilt messaging service; we can message the vendor about the quality and feedback of the product.



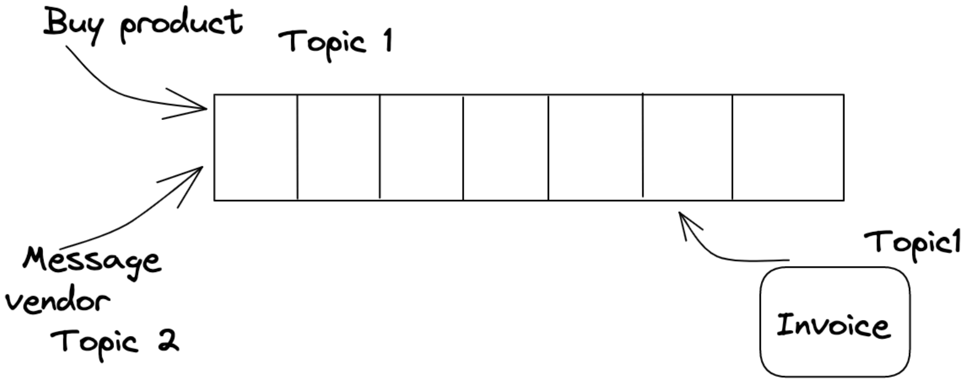
These are the two events, now after that, we want certain things to happen,



Here both the events are very different.
If we publish both of the events in a single persistent queue, and let's say invoice generation has subscribed to the queue then it will get a lot of garbage.



Hence we say all the events are not the same, and we classify them in different topics.

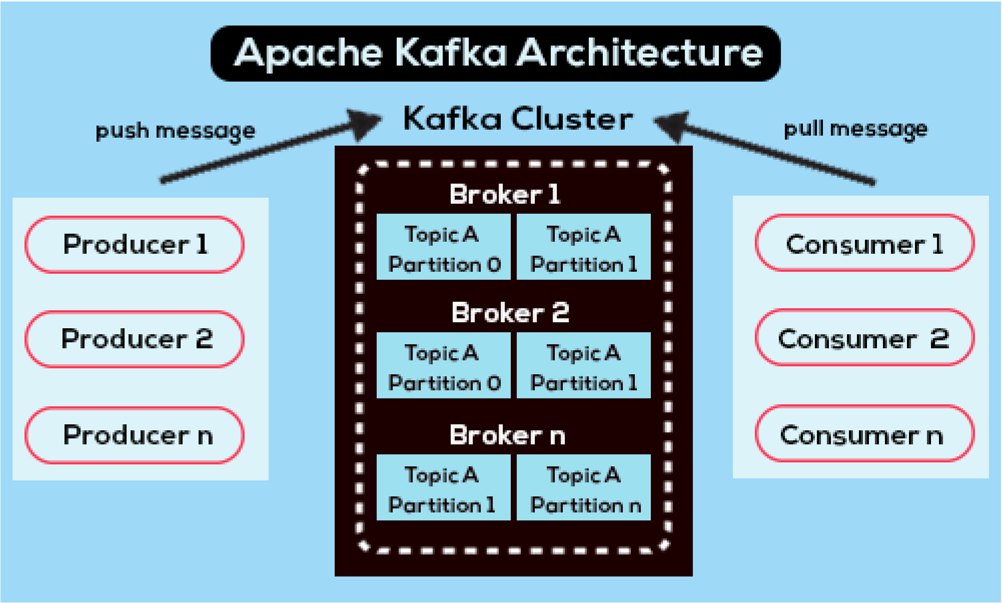


Now the invoice generation has only subscribed to Topic1 and would only get messages from Topic1.

One such high throughput system that implements persistent queue and supports Topics is **KAFKA.**

In general, persistent queues help handle systems where producers consume at a different rate and there are multiple consumers who consume at a different pace asynchronously. Persistent Queues take guarantee of not loosing the event (within a retention period) and let consumers work asynchronously without blocking the producers for their core job.

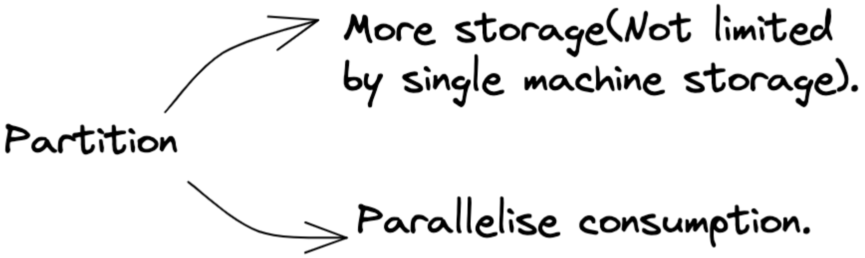
Kafka:



Terminologies:

- **Producer:** Systems that publish events (to a topic) are called producers. There could be multiple producers.
- **Consumer:** Systems that consume events from subscribed topic (/topics) are called consumers.
- **Broker:** All machines within the Kafka cluster are called brokers. Just a fancy name for machines storing published events for a topic.
- **Partition:** Within a single topic, you can configure multiple partitions. Multiple partitions enables Kafka to internally shard / distribute load effectively. It also helps consumers

consume faster. *More on this later.*



- **Event retention period:** Kafka or any persistent queue is designed to store events transiently and not forever. Hence, you specify event retention period, which is the period till which the event is stored. All events older than retention period are periodically cleaned up.

Problem statements:

- ***What if a topic is so large (so there are so many producers for the topic), that the entire topic (even for the retention period) might not fit in a single machine. How do you shard?***

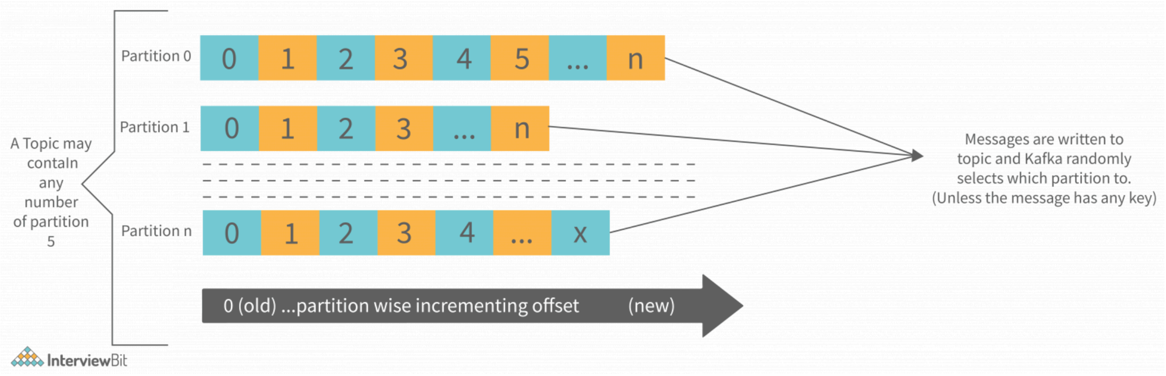
Kafka lets you specify number of partitions for every topic. A single partition cannot be broken down between machines, but different partitions can reside on different machines.
Adding enough partitions would let Kafka internally assign topic+partition to different machines.

- ***With different partitions, it won't remain a queue anymore. I mean wouldn't it become really hard to guarantee ordering of messages between partitions?***
    - For example, for topic messages, m1 comes to partition1, m2 comes to partition2, m3 comes to partition 2, m4 comes to partition 2 and m5 comes to partition 1. Now, if I am a consumer, I have no way of knowing which partition has the next most recent message.

Adding ways for you to know ordering of messages between partitions is an additional overhead and not good for the throughput. It is possible you don't even care about the strict ordering of messages.
Let's take an example. Take the case of Flipkart. Imagine we have a topic Messages where every message from customer to vendor gets published. Imagine we have a consumer which notifies the vendors.
Now, I don't really care about the ordering of messages for 2 different users, but I might care about the ordering of messages for the messages from the same user. If not in order, the messages might not make sense.

What if there was a way of making sure all messages from the same user ends up in the same partition. Kafka allows that.
Producers can optionally specify a key along with the message being sent to the topic.
And then Kafka simply does hash(key) % num_partitions to send this message to one of the

partition. If 2 messages have the same key, they end up on the same partition. So, if we use sender_id as the key with all messages published, it would guarantee that all messages with the same sender end up on the same partition. Within the same partition, it’s easier to maintain ordering.

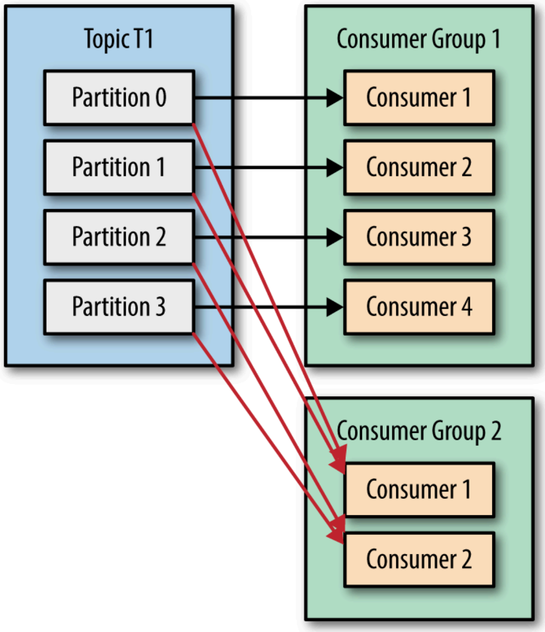

Partition 0
0 1 2 3 4 5 ... n
Partition 1
0 1 2 3 ... n
Partition n
0 1 2 3 4 ... x
A Topic may contaln any number of partition 5
Messages are written to topic and Kafka randomly selects which partition to. (Unless the message has any key)
0 (old) ...partition wise incrementing offset (new)
InterviewBit

- **What if a topic is super big. And hence it would take ages for a single consumer to consume all events. What do you do then?**

In such a case, the only way is to have multiple consumers working in parallel working on different set of events.

Kafka enables that through consumer groups.

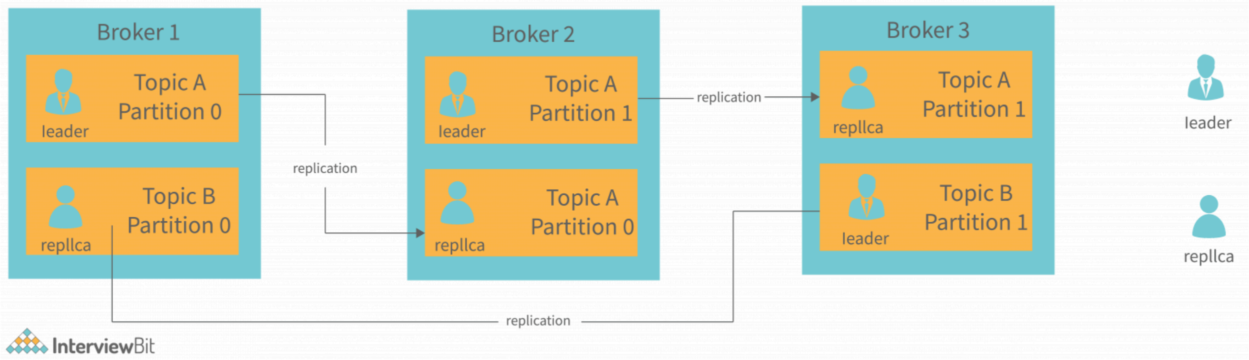
A consumer group is a collection of consumer machines, which would consume from the same topic. Internally, every consumer in the consumer groups gets tagged to one or more partition exclusively (this means it's useless to have more consumer machines than the number of partition) and every consumer then only gets messages from the relevant partition. This helps process events from topics in parallel across various consumers in consumer group.



- **If one or more machines(broker) within Kafka die, how do I ensure I never lose events?**

Same solution as every other case. **Replicate**.
Kafka lets you configure how many replica you wish to have. Later, for every partition, primary replica and other replicas are assigned between machines/brokers.

Example: *See below image, when Kafka has 3 machines, 2 topics and each topic has 2 partitions. Replication configured to be 2.*



- **If I am a producer or a consumer, how do I know which Kafka machine to talk to?**

Kafka says it does not matter. Talk to any machine/broker in Kafka cluster and it will redirect to the right machines internally. Super simple.

- **If I am a consumer group, and I have already consumed all events in a topic, then when new events arrive, how do I ensure I only pull the new events in the topic?**

If you think about it, you would need to track an offset (how much have I already read) so that you can fetch only events post the offset.
More reading: https://dattell.com/data-architecture-blog/understanding-kafka-consumer-offset/

References

Kafka

1. https://kafka.apache.org/documentation/#design
2. https://blogsarchive.apache.org/kafka/entry/apache-kafka-supports-more-partitions
3. https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/

4. https://www.conduktor.io/kafka/kafka-topics-choosing-the-replication-factor-and-partitions-count/
5. https://www.conduktor.io/kafka/kafka-fundamentals/

Zookeeper

1. Zookeeper design goals: https://zookeeper.apache.org/doc/current/zookeeperOver.html#sc_designGoals
2. Observer pattern in Zookeeper
   a. https://apache.googlesource.com/zookeeper/+/3d2e0d91fb2f266b32da889da53aa9a0e59e94b2/src/java/main/org/apache/zookeeper/server/ObserverBean.java
   b. https://apache.googlesource.com/zookeeper/+/3d2e0d91fb2f266b32da889da53aa9a0e59e94b2/src/java/main/org/apache/zookeeper/server/WatchManager.java
3. File structure in Zookeeper
   a. https://zookeeper.apache.org/doc/r3.9.1/zookeeperOver.html#sc_dataModelNameSpace
   b. https://apache.googlesource.com/zookeeper/+/3d2e0d91fb2f266b32da889da53aa9a0e59e94b2/src/java/main/org/apache/zookeeper/server/DataNode.java
   c. https://apache.googlesource.com/zookeeper/+/3d2e0d91fb2f266b32da889da53aa9a0e59e94b2/src/java/main/org/apache/zookeeper/server/DataTree.java