Every node is an Broker.

Controller creates Leader and leader creates it Group of brokers (Followers)

If Leader is not able to create then new Leader is chosed.

Zookerper keps all the clusters metadata.

For every single cluster there will be one Zookeper.

First look into Zookeper and start it.

Next start the broker it registers Zookeper.

Zookeper also responsible for creating Controller, and get info when Leader is create/Followers

Zookeper is like NameNode

After all information is received by zookeeper, then all the details will be broadcasted to all nodes.

Zookeper itself can be scaled.

Its scala based system.

Controller Responsibility-> Health of every node, Risk Policy/Task Redundancy,

One broker can post any number of topic, then another topic will be created.

Manager using gossip Protocol -> To choose who should be the leader.

Typically the oldest broker will be made the Leader.

Zookeper-> Distributed Configuration Management,

Consensus building through coordination and locks

Manage cluster Metadata

Stores data in data tree structure

Kepps all the servers in synch using Atomic Broadcast

Messaging Structure and Processing, 3 info

{TimeStamp}, {id}, {DataContent}

Within a topic-> It’s a list with ordered based on timestamp

All the data is Immutable for topic, because of high throughput

Topic list

0 1 2 3 4

Reading data:

Subscribe and offset (0->1->2)

Objects are immutable, and any message published cannot be rechanged. It should be handled by client.

Queue-> 1 to 1 communication( 1 – 1 (First who picks will read others will not get even though we have many consumer))

JMS, AMQP for pull message and topic.

RabbitMQ-> 1 consumer registered to it, first created queue and gets the data.

When another consumer tries to get again, then internally will create another queue and push data

If 3 customers then 3 queue.

If we subscribe later then only the data at what time he signs up get only that data.

Slow consumer-> Rate at which consumer is receiving will be low then lot of queue.

Queue-single queue.

Topic- Multiple queue.

If we have 2 subscribers one is fast and other is slow. So the second queue will be have large queue and its affects perf.

Kafka:

Per topic we can define the retention time in Kafka.

Customer can also tell from what offset you want you get it from there.

Defining the replication we have to do at topic creation:

Partitioning-> Scalability in per topic how many messages per minute and retaining with 7 days of retention. So we cannot store in single log file, So we can partition with Datewise,Count wise.

Partitioning-> Allows us to split the data. As the no of data increases we can increase the Partitioning.

So because if requirement changes for more message then we cannot create new message, because lot of application can be hooked up. So we cannot do that.

Instead we add new nodes and increase Partion.

When one producer pushes data then data will be portioned to Partition 0,1,2 and also they have timestamp.

Message 1 to partion 0, another to Partition 1 and another to partition 2, usually

Topic can reside in multiple brokers.

1 partition we be in one broker.

**There will be one leader per topic partition. For scaling I can change no of Partition which will result in more leader and Followers**

All partition are independently managing independent Commit Log.

We can tell I want to produce to one partition.

We cannot increase topic scalability without partitioning.

* First time during creating topic we ask zookeeper to create 3 partition.
* When producer wants to produce data it need not tell zookeeper as the data is broadcasted using atomic propogation.
* Producer will send data to 3 ip because any one will be up, since zookeeper keeps in sync with metadata. Then we get to know where we have partion1 for topic1.
* Producer will directly publish to it.
* Message processing happens just on Leader node.

Kafka has its own custom protocol, an binary level on top of TCP.

If we push everything on kafka broker then performance will hit, for that we have producer and consumer client.

We have Bootstrap Servers-> When producer starts which all ip of 2-3 ip of leader.

When data is sent on server side, we can define what kind of serialization we need to use.

So that how data is published and how consumer will deserialize it.

ProducerRecord:

What is topicname and what value, partition, Timestamp,key

If we want to control our timestamp instead of Kafka adding it.

Key tells like when one message should go to which partition.

But by defining it. It limits the scalability.

If my app I send hostname for key, then I get same partition. If new partition added then new key we create so all goes to new key.

All message we want to log will go to same node.

Direct if we pass in the ProducerRecord with Partition(HardCoded not recomonded) [0-3].

If default then it goes to Round Robin Partioning Stratergies.

If Key is given, then uses hashing algorithm Key[Hash]. It should be valid business case and we should not put lot of load on single partition .

Key to identify which partition to use dynamically, find how many partition are available for the Key. So increasing the partition will not hamper the scalability.

Every Topic-> {size, time} we can set threshold like size, or time whichever is first it pushes.

It internally creates a buffer.

Everything about Partitioning happens on Message Producer end.

Message Consumer:

Consumer should also know from where I should pull message similar to producer.

So it ask bootstrap which ip I should use for getting value.

Fetcher will keep buffering the record.

Consumer runs in single thread.

Kafka is always Push model.

Here consumer will always so and pool for extra messages.

Consumer has subscribe method per topic level, So it looks and finds which partition has that topic and pools from all partition.

Consumer

If we have to order based on time based, client needs to handle it as the data can be received at different time.

Other scenario like sending mail on receiving value so time based is not required.

Consumer has to manage the offset. So for second polling will pick from next offset.

Consumer Offset topic will store the offset(Commited offset). Like commit logs saying acknowledge.

Consumer.commit()-> stores last offset position and next time it picks next offset.

ConsumerGroup-> Since consumer runs of single threaded, because we don’t want to read the same message again. For this we have Consumer Group-> For every consumer we have GroupId. Based on this it will not read again.

ConsumerCoordinatior-> It will balance what ever request consumer gets. It handles distributed offset management as well.

If Logs we can see multiple offset files.

Commit Log-> On creating topic we see details in commit log.

Kafks-server-start.sh->Starting broker

Zookeeper-server-start.sh-> Starting zookeeper.

Built in zookeeper will already be available.

Zookeeper, broker1, broker2,

List topics-> We see consumer-offset topic aswell

By default kafka maintains 50 partition of it.

Each partitiontells-> topic, partitionno, who is the leader,

TO check go to filesystem-> temp->consumer\_offset\_1

Consumer --From-beginning

By default based on timing it picks

Consumber.subscribe(listOfTopics) // We can subscribe to multiple topic

CommitSync: OffsetPosition if it fails the system will recommit

CommitAsync: If commit fails it will not recommit, but performance is improved.

New Brooker(‘MultipleIPS’);