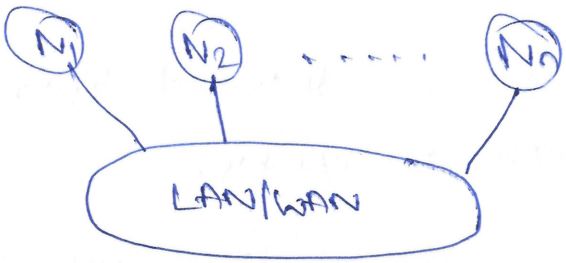
Distributed Systems.

A Collection of independent computers that appear to its users as one computer.

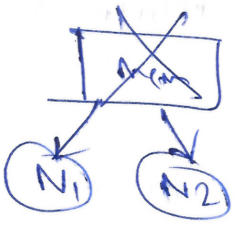


→ Nodes connected using LAN / WAN

LAN implemented – Fiber optic, Co-axial cable

WAN – Satellite Communication.

→ No Physical memory shared b/w the Nodes. So only way to communicate is by sending messages on LAN



→ Message communication time is significantly larger than the event computation which happens on a single node to do some meaningful processing. $(t_m >> t_e)$

* Cluster – A single rack in a Data Center having a group of machines. Even a cluster is a distributed system, because processors are blazingly fast and hence event computation time has shrunk, but message communication b/w nodes is not so fast.

CAP Theorem: Consistency, Availability and Partition Tolerence

→ Consistency – Always give me back the most updated / recent record.

→ Availability – Always available to serve requests.

→ Partition Tolerance – Nodes cut off from others.

Ex: Power cut off in a rack of Data Center
Data Center down etc

* With distributed systems all three is not possible. i.e CP, CA or AP is possible

* Partition Tolerance is a must for a distributed system. So make a decision on consistent or available.

Ex: Coffee Shop Shared Writing Project.

* CP System – When a request comes over, since you don't have the recent updated record, you don't give anything.

* AP System – Give an update, which was may be a minute older and let the other for handle the rest.

Used by Uber.

⇒ All three things of the CAP Theorem is not possible.

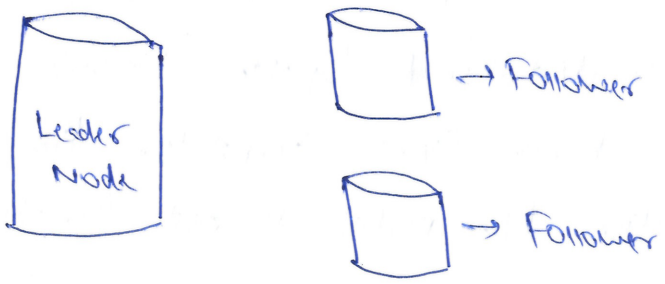
Examples of Distributed System :-

Amazon.com – Tons and thousands of systems working together to seem like one system.

Cassandra Database – Distributed key-value store.

Read Replication :

Ans : ?



Cons.

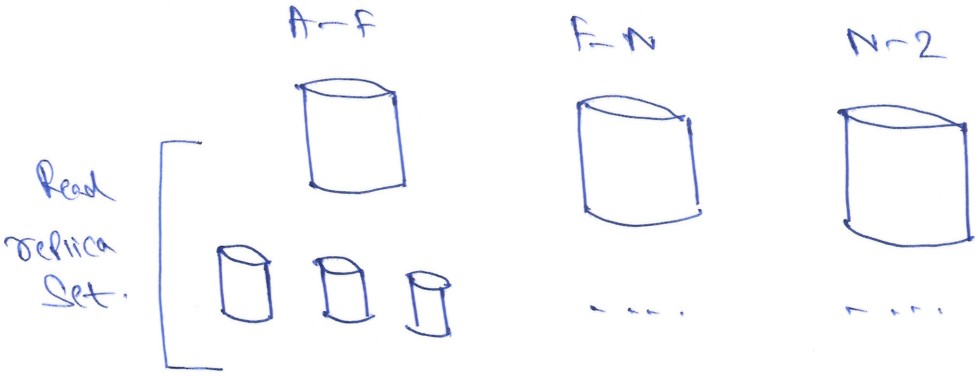
① Complexity – Copy to other nodes.

② Consistency – All 3 nodes not on same page. So eventual consistency instead of strong consistency

Advantages : More reads than write kind of application. i.e Read Heavy.

Sharding Distributed Systems:

* Read replication is not working for you. Because lot of time spent on replicating the nodes. Also write heavy
* Sharding - helps to split the load.



* Now we tripled the write capacity.

⇒ Mongo DB uses this strategy.

Here key - shard mapping is now required

Cons:
* limited data model - we some how need to have a key-value or shard key.
* Complex.
* limited data access - Not good for analytics. Then it will be too much scatter-gather

---

Distributed System Cluster:

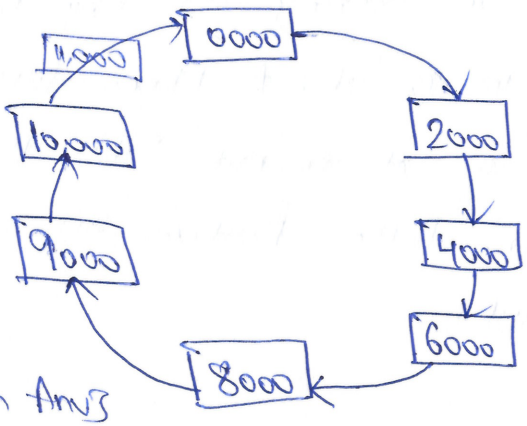
Why?

* Servers are fault-tolerant and they provide seamless access to a service.
* Any server on the LAN can respond to requests.

Adv:
* No single point of failure
* Provides scalability
* Reliable design
* data shared b/w all machines.

Consistent Hashing – Distributed session. (or) Distributed Hash Table.

Anu : Coffee
⇓ Hash the key
9F72 : Coffee



1/8th of traffic distributed uniformly.

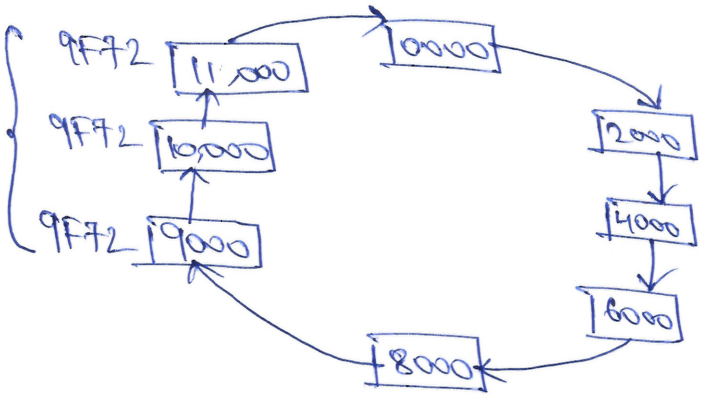
* After some time when Anu comes back & asks just hash the key again to locate the item.

But what happens when the node 9000 goes down, like in Products like Cassandra, Dynamo DB etc. Scale is not a Problem in Consistent Hashing.

* One Solution is replication.

Replication.

* All gets the same Data no master/slave.



* Above creates a Problem of Consistency. What if one of the Nodes did not get updated? or may be not available so they dint get the update. How many replica's should have all of the data.

This is when Eventual Consistency comes into Picture. We need to take a call on Consistency. (or) have Strong Consistency and compromise on little bit of Availability. i.e Some latency.

* When to use this kind of a DB (Distributed like Dynamo, Cassandra)
  - Scale (i.e large business)
  - Transactional Data. (Business transactions) i.e Data which is changing a lot.
  - Always on. Highly available.

* Check existance of data in a node using a Bloom Filter.

Bloom Filter is an extremly fast way to test the existence of data structure in a set. A bloom filter lets us know if an item exists in a set or not. This is a good way of avoiding unnecessary I/O. False positives are possible but false negative are not.

Messaging. (Pub/Sub)

→ Means of loosley coupling Sub-Systems.

→ Messages consumed by subscribers.

→ Created by one or more Producers.

Example: Coffee Shop: Coffee orders in CCD lined up if lot of people are there. Single Queue with some Queue. (Similar to enterprise Queue like MQ.)

→ Stream of messages organized into topics.

→ Processed by brokers. (i.e a Server doing message processing)

→ Persistent over short term. (like a broker is broken for short time).

Messaging Problems?

* What if a topic gets too big for one computer? Topic is a Queue and if some of it is in one System and other in an other System then how do we maintain ordering. Because machines operating concurrently.

* What if one computer is not reliable enough? Node failures.

* How can we guarantee delivery?

Kafka. [Written in Scala, Used by LinkedIn, Uber, Netflix]

- Distributed message Queue.
- Horizontally Scalable, fault-tolerant
- Durable
- 100's MB reads and writes/seconds.
- Thousand of clients per broker.

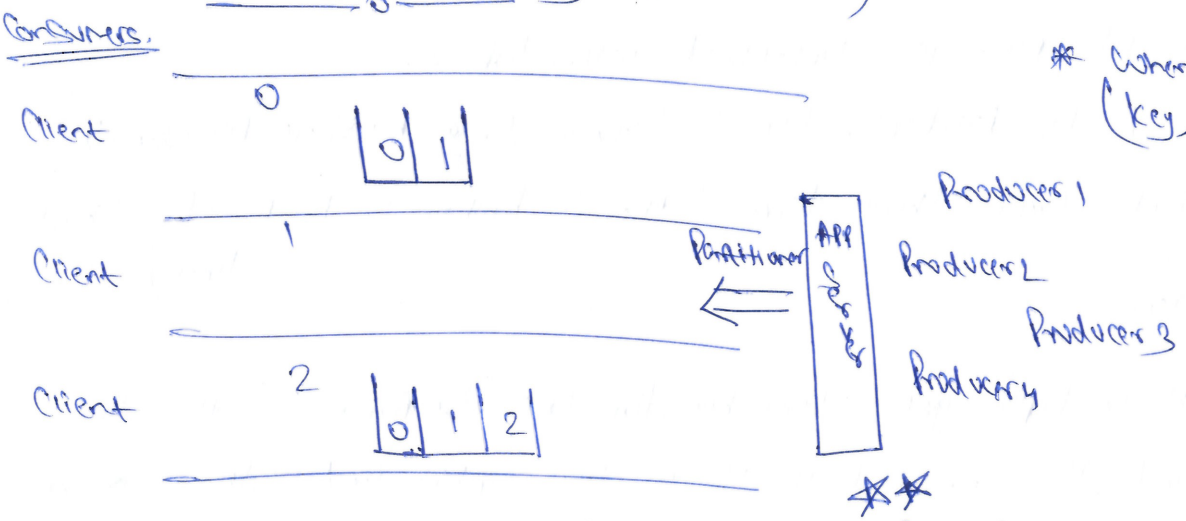
] Fast !!

Definitions.

* Message : Immutable array of bytes.
* Topic : a feed of messages
* Broker : one of the servers of a cluster.

What if the topic gets too heavy load. i.e Some kind of web activity tracking i.e Linkedin. or a log processor.

Topic level Partitioning. (Same Topic)



* When Sending a message (key, body)

** Indexing at Partition level. Per Partition Ordering.

⇒ Producer opening a web socket to a machine which has a Partition 0 of a topic or Partition 1 of a topic. All of them able to generate a Push messages in a very fast / high speed.

⇒ Consumers then read messages of individual Partitions, in an ordered fashion. There is no global ordering available

* A Scaling Strategy for a topic
* Done in the Producer by a Pluggable class and a message key.

Apache Zookeeper. (highly reliable distributed co-ordination)

* Required part of kafka architecture.
* Producers use it to find partitions and replication information. i.e what are the topics, what replication do they have etc
* Consumers use it to track current index. Next message entry point for a consumer is kept track by zookeeper.

Replication:

* Brokers can fail.
* Configurable in-sync replica set.
* One leader, n-1 followers.
* Only leader communicates with the client. ie no slaves talk to the client.
* When the leader fails, a new leader is elected. Hence consistency guaranteed.

Configurable → what happens when a leader failed and a write request comes in and the election process for leader is on-going.

Apache Kafka

→ Pub/Sub based model.

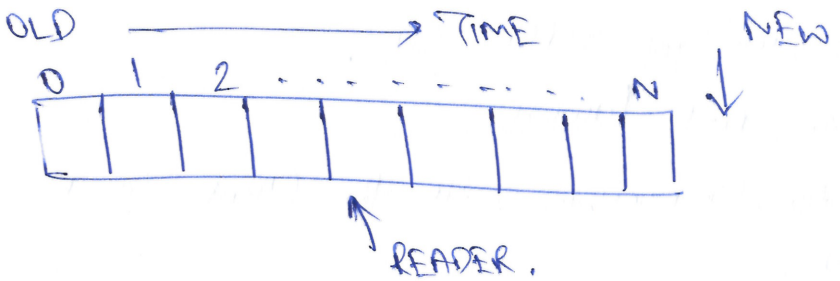
① Publish & Subscribe – to streams of data like a messaging system.

② Process – streams of data efficiently in real time

③ Store – streams of data safely in a distributed replicated cluster.

→ Real Time Stream Processing.

→ Horizontally scalable, fault tolerant.



Above is a stream which can have a web page activity details or a log processor. Kafka puts each of the above topics as a partition allowing it to put in a different cluster and have fast processing.

* Write to a distributed database

How do we maintain consistency of data?

→ Create replica sets. – To avoid sequence of failures.

Much like mongoDB, it has a central controller and has a election process on failure. This is acheived using Zookeeper.

i.e maintain who is leader – writes

who is consuming what