**Design a key value store**

Remember this is not the same

Functional vs non-functional requirement?

High level and then low level here

Do a high level design

**There are 2 sections to this**

**1.** The first one is from byte byte go

2. The 2nd one from youtubeu

Part 1 from byte byte go

**What happens when you choose consistency over availability here?**

If we choose consistency over availability (CP system), we must block all write operations to *n1* and *n2* to avoid data inconsistency among these three servers, which makes the system unavailable. Bank systems usually have extremely high consistent requirements. For example, it is crucial for a bank system to display the most up-to-date balance info. If inconsistency occurs due to a network partition, the bank system returns an error before the inconsistency is resolved.

However, if we choose availability over consistency (AP system), the system keeps accepting reads, even though it might return stale data. For writes, *n1* and *n2* will keep accepting writes, and data will be synced to *n3* when the network partition is resolved.

xs

Using consistent hashing to avoid the node added serviers issue:

* First, servers are placed on a hash ring. In Figure 4, eight servers, represented by *s0, s1, …, s7*, are placed on the hash ring.
* Next, a key is hashed onto the same ring, and it is stored on the first server encountered while moving in the clockwise direction. For instance, *key0* is stored in *s1* using this logic.

For example key 0 is mapped to s1, s2 and s3 here.

Consistency here

**How to ensure the write data operation is successful?**

**A diagram of a diagram

Description automatically generated**

**In the above example**

1. Data is replciated at *s0*, *s1,* and *s2*. *W = 1* means that the coordinator must receive at least one acknowledgment before the write operation is considered as successful.

A write quorum of size *W*. For a write operation to be considered as successful, write operation must be acknowledged from *W* replicas.

**How to define eventualy consistency?**

Eventual consistency: this is a specific form of weak consistency. Given enough time, all updates are propagated, and all replicas are consistent.

**How to resolve inconsistencies among replicas. Versioning and vector locks are used to solve inconsistency problems?**

**How does conflict reolustion happen here**

As shown in Figure 7, both replica nodes *n1* and *n2* have the same value. Let us call this value the original *value. Server 1* and *server 2* get the same value for *get(“name”)* operation.

A diagram of a cylinder

Description automatically generated

Figure 7

Next, *server 1* changes the name to “johnSanFrancisco”, and *server 2* changes the name to “johnNewYork” as shown in Figure 8. These two changes are performed simultaneously. Now, we have conflicting values, called versions *v1* and *v2*.

**A diagram of a diagram

Description automatically generated**

**Technique # 1 here**

1. There is sth called the last write wins?

2. Technique # 2

**Use vector clock versioning here**

A vector clock is a *[server, version]* pair associated with a data item. It can be used to check if one version precedes, succeeds, or in conflict with others.

The conflict can be handled by the client here

**Handling failures**

As with any large system at scale, failures are not only inevitable but common. Handling failure scenarios is very important. In this section, we first introduce techniques to detect failures. Then, we go over common failure resolution strategies.

Using the gossip protocol:

A screenshot of a computer

Description automatically generated

And then here we have the code

1. Node *s0* maintains a node membership list shown on the left side.
2. Node *s0* notices that node s2’s (member ID = 2) heartbeat counter has not increased for a long time.
3. Node *s0* sends heartbeats that include *s2*’s info to a set of random nodes. Once other nodes confirm that *s2*’s heartbeat counter has not been updated for a long time, node *s2* is marked down, and this information is propagated to other nodes.

What’s a coordinator here?

A coordinator is a node that acts as a proxy between the client and the key-value store

**This part following is from the video**

https://www.youtube.com/watch?v=6fOoXT1HYxk&t=668s

**What's the idea of a key-value store?**

1. You are basically designing a cache system like redis here, the guy in the video is from Microsfot

1. The ask is design a memcache type system

**Non-fucntiona nd functinal requirement here**

1. And then so here we have the code,

Functional req:

put and get here

putting the key and value here,

**What are non-functional req here?**

1. Avalialibity here and then highly available here, and if system is higlyy available

2. Remember a

Steps:

Start with functional req:

1.

A person wearing headphones

Description automatically generated

**To maintain a hash table here**

1. Using a hashtable for fast retrieval here, Service A will check cache, if not then update the cache

2. What's con of hash table? what's a better approach?

Expensive memory, fix: take cache policy into account here, use lru cache instead here

3. That's better here

**Why choose LRU over hashmap?**

how do you track which item was used and when

right because has table only provides you key value paay storage but it does not provide you any mechanism to track

which was stored and which was accessed and when just before we move on are

What data structure is then used here?

1. Using a doubly linkedlist and a hash table here, as data enters into the q, change

2. Any item newly added move to the head of the linkedlist, point the head to the first node here

3. If full then remove the last element which is the LRU in this case.

4. Will work with the cache table here and then way more system cache and system here

Alphabet A - Z

**Why use multiple servers here?**

For huge applications, like large scale key-value stores, it’s infeasible to store the whole data on the one server. To understand this phenomena more concretely, think of a case that our application uses relational database and, that some of our tables become so large(contain so many rows) that it eventually becomes extremely slow or even worse, we simply get out of disk space to store ever increasing amounts of data ( Yes! these kind of problems exist in huge applications).

For example

Service A, Service B the following:

Cache A : A -M stores key A -M here

Cache B: N - Z here

Can have cache for different hosts here

A diagram of a service

Description automatically generated

These are deployed on different hosts, 1 host to maintain on service A '

WHile designing these caches

**What's some of the challenge with using a hash functino approach here? What's the alternative**

Using a hash\_function (key) to return the # of hosts here:

Always mod the # of hosts here, 3 caches hash\_function(key) & number of hosts that you have

1. Challenge with this approach: 3 cache for example. If u add another cache server tomorrow what happens?

okay so this is KN approach and the reason I called it naive approach it does work very well but there is a challenge with this approach if something happens to any of the caches assuming you have three cach or take an example for this case since we are talking about two cach assuming you have two cach and tomorrow if you're adding a cach c or if something goes something happens to cash a and cash B and they go down for any reason the M the hash function will start returning the wrong value and the reason is even though

it says number of host which remains with changes right and so you don't get the same answer so if I change the number of host I will don't get the same answer I got the previously when I added the cache because when I added the cache the number of host was two so function returns X but now one server goes down another server comes up have three host or four host so the function does not return the same value so I I'll go and I'll check the key in the server where it does not even exist yeah so how do we tackle this one simple way to solve this problem is consistent hashing

Basically the # of hosts change the hash fxn no longer works here, key for the server is now gone.

**Solution**: consistent hashing here,

A circle of A-C (key for each server) and now the main thing

Basically instead of request it will be cache coming into here, nothing gets changed for A and B,

If cache D is added now

A -> B -> C -> D (think of the above as a ring)

Only the keys in C will get affected here.

Basically using consistent hashing only subsets of keys get affected here not everything as said previously.

**Who does the caching mechanism?**

1. Build a cache client, which will be deployed with each server you have here and there.

**2. How does cache client get the name of the server here?**

mind is that how does cash client gets the name of the cash server right so we haven't talked about that part part as of now so let's jump into it there are multiple ways

you can always share let's say I have two server cach a and cas cash p and they have specific URL and we know that we tomorrow we can add more caching so how does cash client gets to know the URLs right

**Configuraiton files**

one naive approach you can always take is have a configuration file which lists down the URLs for each caching server and deploy them as part of Ci or devops process right one

**Polling the file names from global location**

other approach you can always take is that the services all the Clash or the cach clients fetch these files periodically from a Global location or you can think of an S3 file right but there is a drawback to it you cannot poll too often to the S3 server just to get the list of the uh caching server you have right one better approach

**Using zookeeper here**

you can always take is that you can use an intermediate service like zookeeper where whenever a cash server gets registered uh it can add a registry to zookeeper and it's more of an health check and cash client works with the Zookeeper to get the list of an updated cash server So based on the scenario U if you know that cash server cash server may not go down or probably it's not that important you can go for the very first approach naive approach which is simple enough but has an old drawback or you can go for the more advanced approach where you know it's highly critical to have an updated cash server list with us so that we can fetch the right data and also the reason what we want to consider a best approach

here is because if your cash server goes down you are going to have a lot of cash Miss and in that case all the queries will go into database so that can also affect okay now we

**How to improve availiaibilty here in this system?**

To achieve reliability and high availability of the system we need to get rid of single point of failures(SPOF) for each distinct part of our system. One of the most famous solutions to solve SPOF is called “Data replication” on geographlically different system here

Approach 1: Raft Consensus Protocol

All the writes are handled by master node only. Each and every write command is considered successful if and only if the master node gets acknowledgments from majority of followers that this new value has been safely replicated.

Reads can be handled by any one of N follower nodes. Meaning that we can have dynamic number of followers depending on the stress of the system. One quite interesting side effect of replication is that we’ll be able to place replicated geographically closer to the clients that issue bunch of read requests, thus decreasing the actual latency for them.

1. Right now 1 instnace of cache server is runnign rn, can add cache server right now so not an issue

2. How to increase the availability here?

1. Add a read replica here first and then, implement read replicas for caches and ensure high availibity

2.

A screenshot of a computer

Description automatically generated

Get request -> read replica A

**How to make sure read replica gets the most updated data? What's the probelm with the stale data here?**

\*Commands are ordered in chronological order from top to bottom\*See diagram below for better flow visualisation

* (time frame 1) client issues a write(K:V)
* (time frame 1) leader of the cluster gets the request, persists it locally and asks followers to replicate it.
* (time frame 1) majority of cluster’s followers successfully replicate the data which instructs the leader to return to the client “OK” status code, saying that write was successful.
* (time frame 2) client issues a read request read(K) which lands on the follower that hasn’t yet been synced with the rest of the cluster
* (time frame 2) client gets stale/no data as a result of read(K)

**How do we achieve data consistency?**

1. Guarantee if the update stops, all replicas have the same data, how to ensure in the same state here

2. If you have communitive opeartion to update the state here,

**Solution 1: is to fake it first lol**

You just placed an order and want to show a confirmation page to your customer. Because we are trying to read the order right after placing it, we will not see it if there is any delay — replica lagging due to higher load, message broker with a backlog of messages, etc.

A solution would be not to read the orders, and simply show the order information you have in memory at the moment of placing it.

**Solutino 2: use a versinoing here**

If you have a use case where you are making changes to an existing entity and trying to retrieve the updated entity right after, one solution is to define what the expected version of the entity you want to receive is.

 If you do not receive the version you expect or a later one, you know you have to do something and can’t use the information you just received. At this moment you can show a custom message to the user or use a spinner — see the UI Poller approach next — and retry fetching until you succeed.(The poller approach can be better replaced with websocket here)

Another benefit here

Having the version also helps with the concurrency control because at the server you can simply reject a change that expected a version that is older than the one that already exists.

**What's security problem here?**

1.

Part 2 From byte byte go article

**Difference between consistency and availiability**

**Consistency**: consistency means all clients see the same data at the same time no matter which node they connect to.

**Availability:** availability means any client which requests data gets a response even if some of the nodes are down.

**Partition Tolerance:** a partition indicates a communication break between two nodes. Partition tolerance means the system continues to operate despite network partitions.

**What's CAP here and how does it work?**

1. The whole idea is that in case of Network partitions in the system, only one of: Consistency or Availability might be reached. Since distributed systems, by definition are network partitioned, for each and every modern system we need to choose whether to design **AP**(Highly available distributed system) or **CP**(Strongly consistent and distributed system).

network partitioning is **a network failure that causes the members to split into multiple groups such that a member in a** group cannot communicate ...