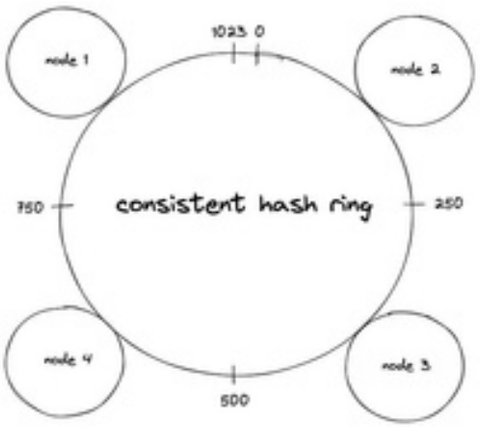
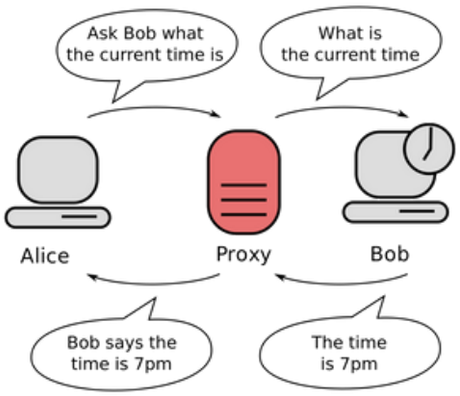




Design WhatsApp (Chat Messaging System), Consistent Hashing, CDN & Proxies

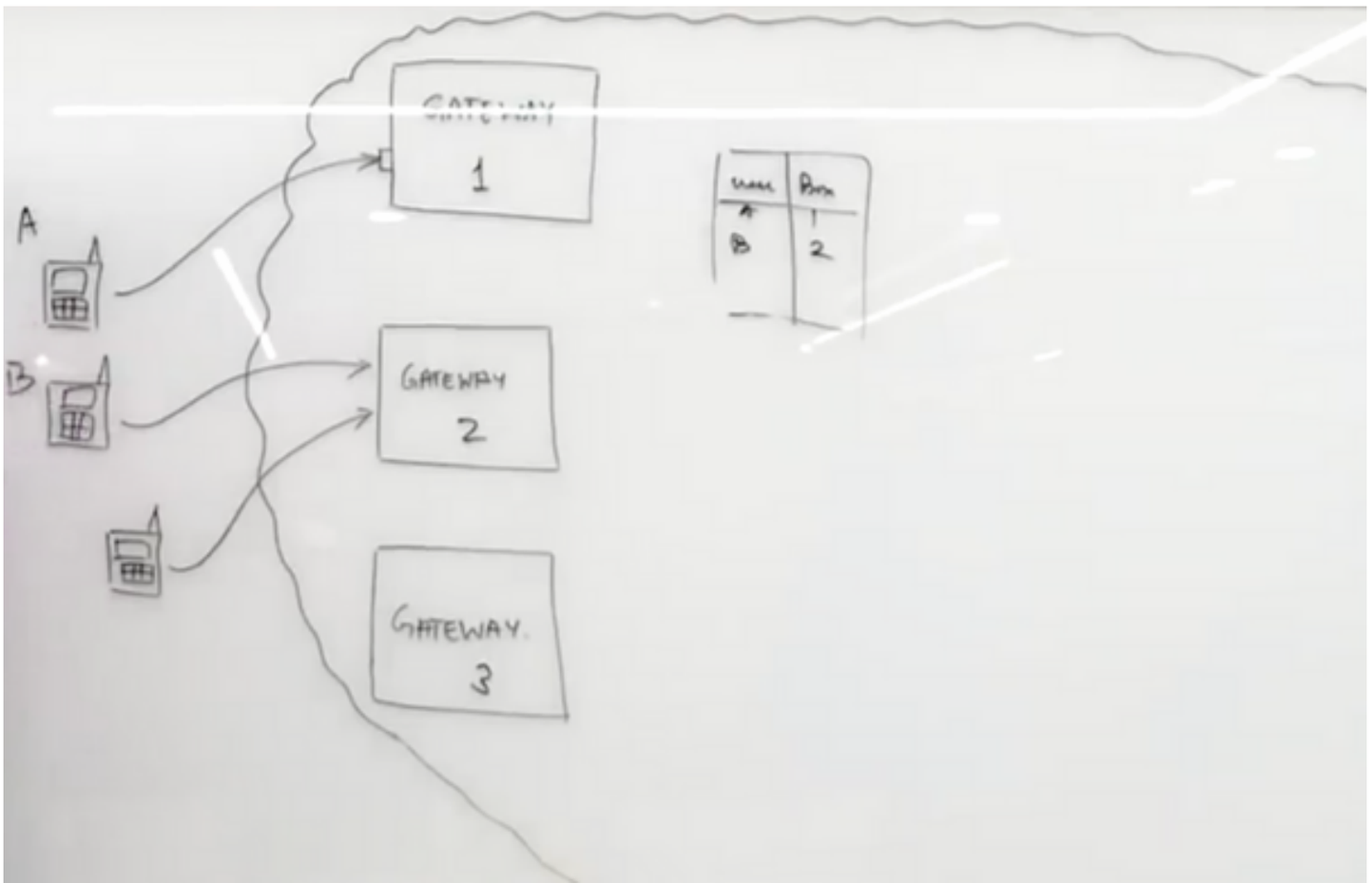
Design WhatsApp (Chat Messaging System)

Key Requirements:

1. One to One Chat Messaging System
2. Online / Last Seen
3. Group Messaging
4. Sent / Delivered / Read Receipts
5. Image / Video Sharing
6. Temporary / Permanent Chats

One to one chat

Single Point Of Failure is important when it comes to whatsapp.



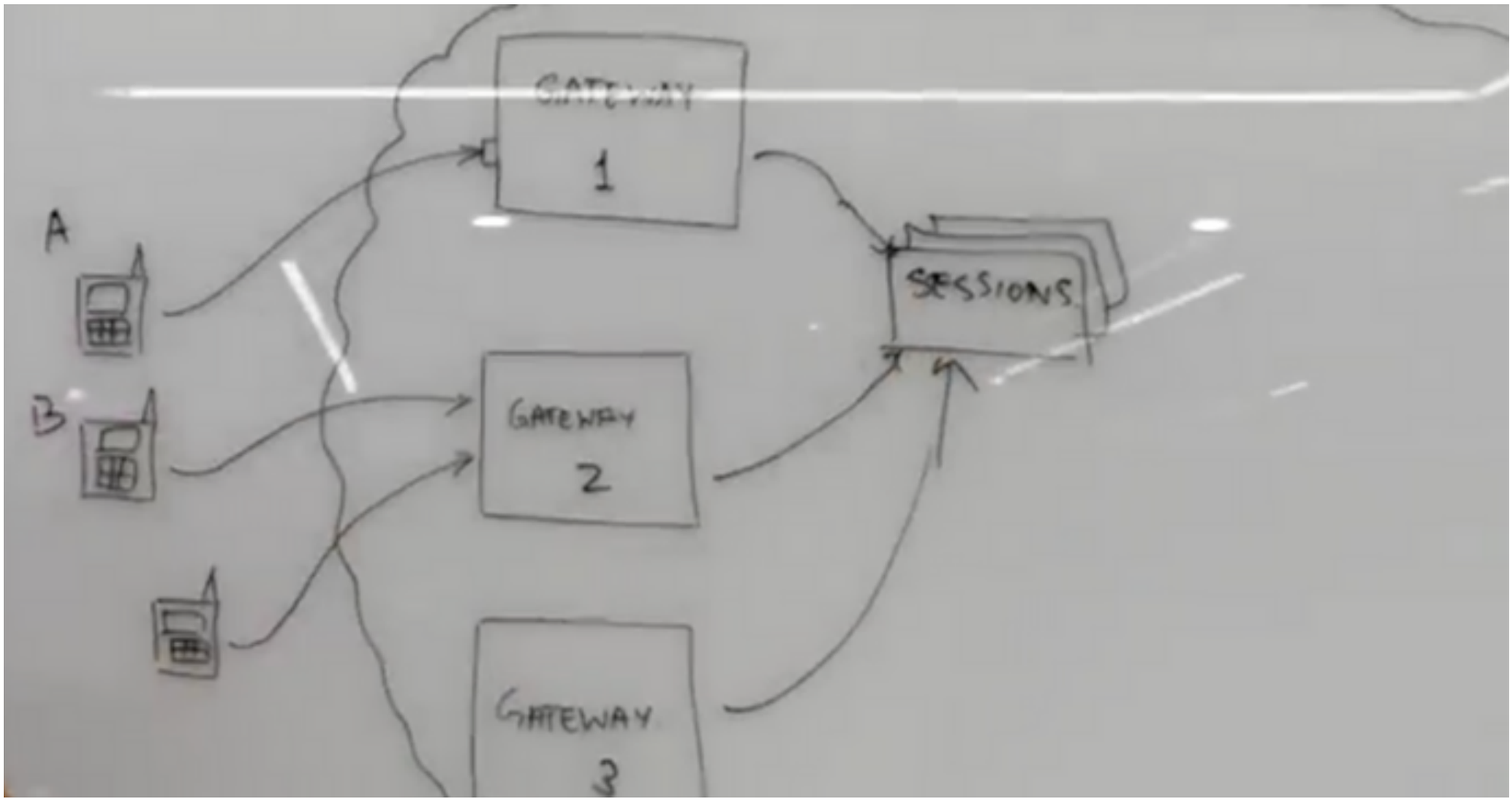
Let's say we want to send messages from user A via gateway 1 to user B via gateway 2. We can have the user to gateway mapping like A to 1.

But this will be expensive as maintaining a TCP connection is itself takes some memory.

Issues:

1. We want to increase the maximum number of connections hence we want to store the mapping in a single box
2. And that information will be duplicated in all three gateway servers via caching mechanism which will be handling it or there can be a database handling this transient information. There can be a lot of updates going on there. There's will be a lot of coupling present at the system.

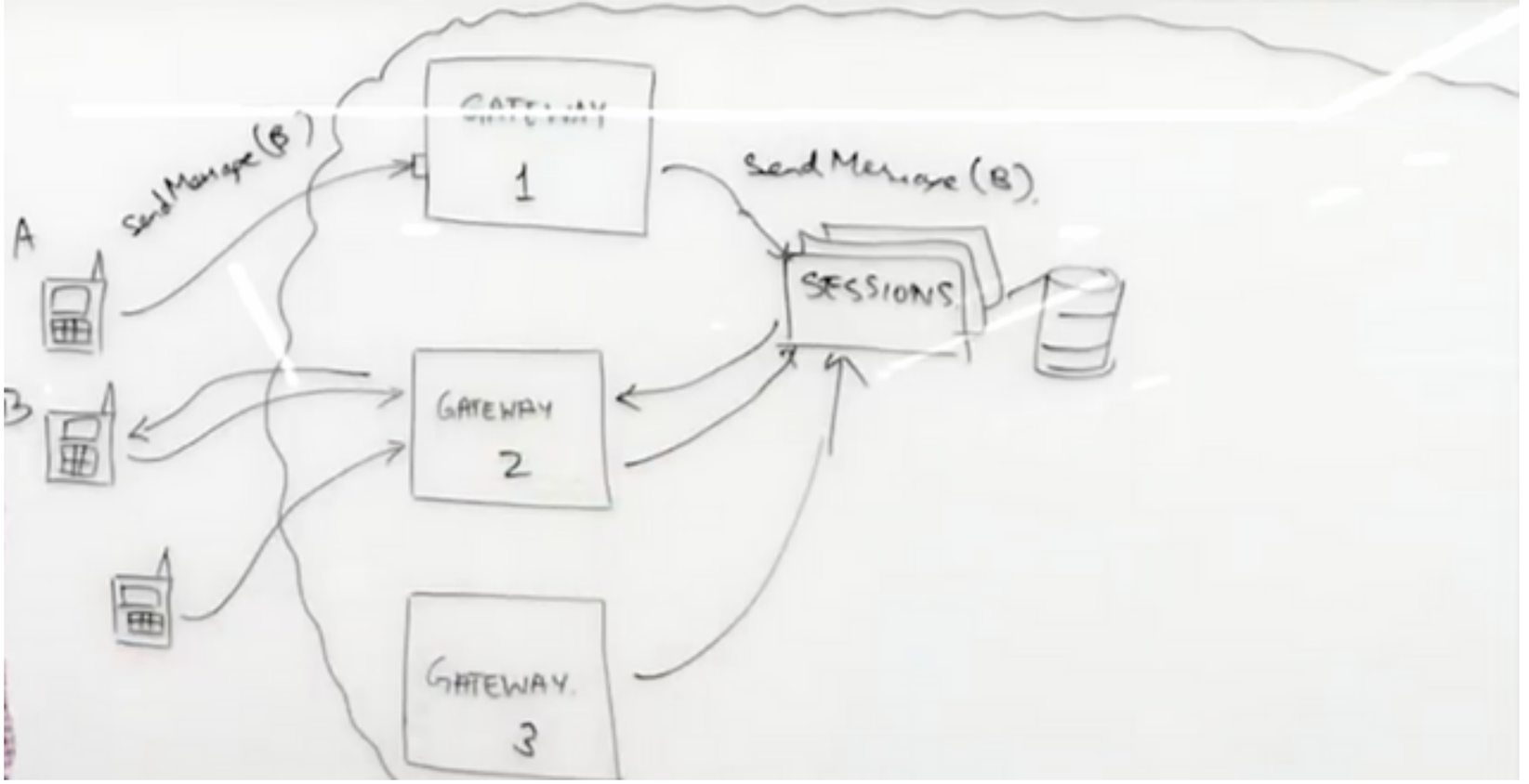
We can have a dumb connection and a session micro-service for storing the connections info.



Session micro-service will have that information via a DB or cache storage. Hence, single point of Failure is covered here.

Gateway will simply send the message from user and the metadata to the session service.

In this way A can send the message to B.



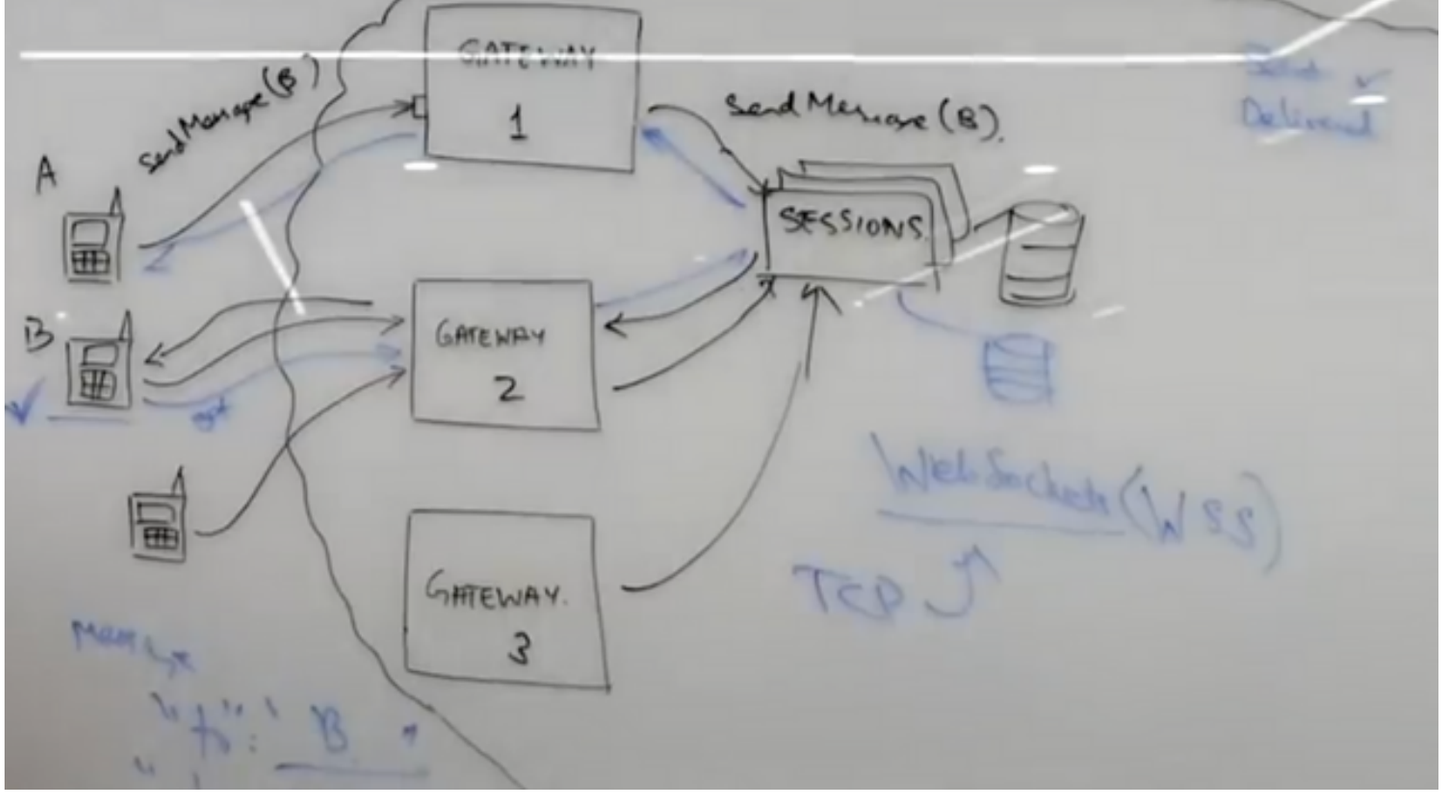
At every minute B can ask the gateway for new messages using long polling.

We can implement this over TCP with the help of websockets (wss). They allow P2P communication, A can send to B and vice versa.

Once the message is sent, user A should be notified that the message has been delivered.

We can have another db for chat messages, once the message is delivered user A will be notified and sent mark will appear with the help of gateway 1 and socket server.

Gateways can be used to update the "received" and "sent" statuses.



3. Online / Last Seen:

Second feature is the last seen or is the person online right now.

Let's say B wants to know when A was online the last time.

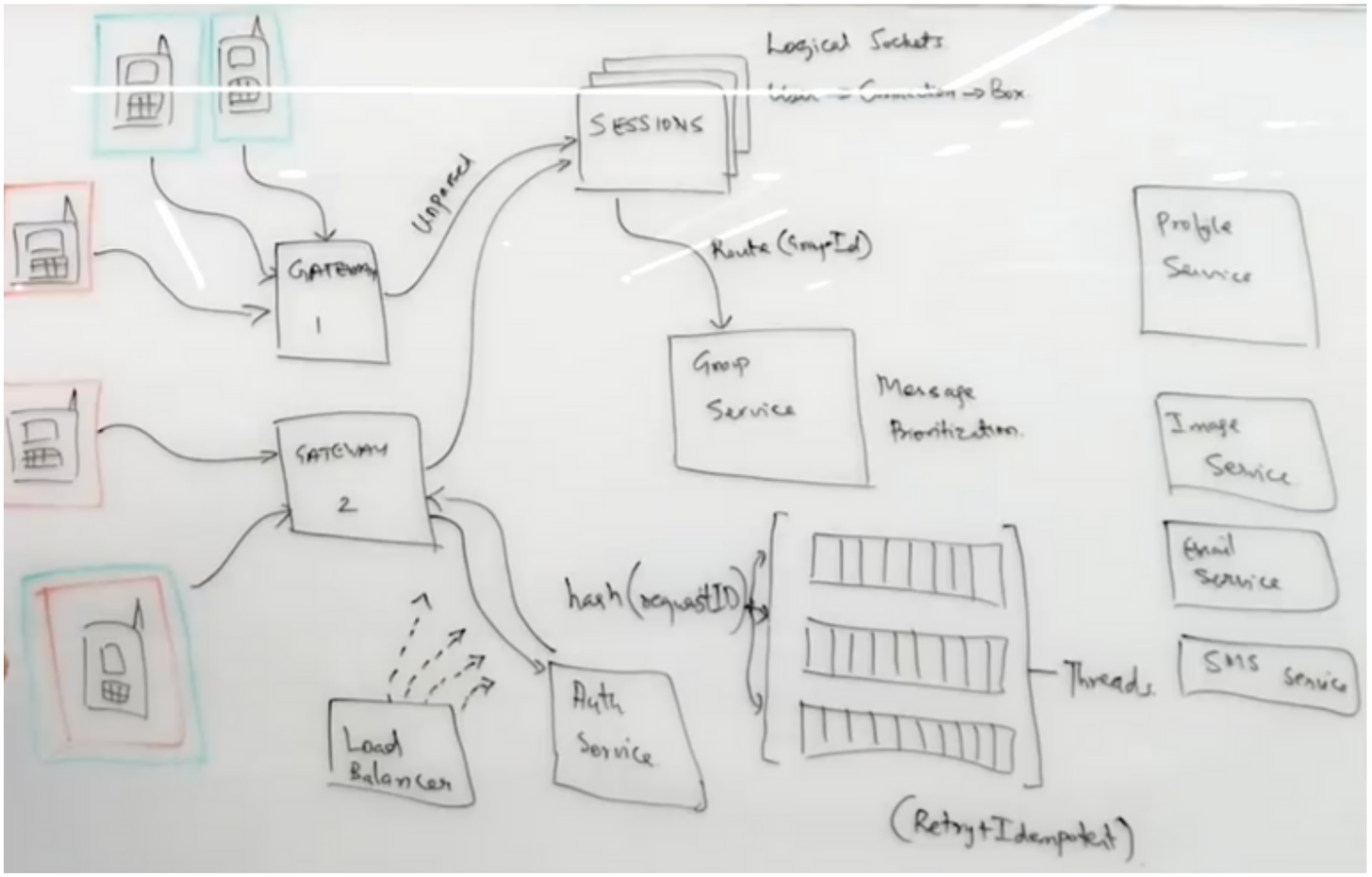
Whenever A does something, last seen timestamp needs to be updated.

We can have a lastSeen micro-service which will do the user activity tracking.

When they send a message to the gateway, lastSeen will be updated.

Two types of messages client will be sending via gateway.
One of those two is User Activity messages and second one is app messages.

And accordingly micro-service will update the db.



4. Group Messaging

It will be really hard for the session service to handle the group messaging logic, that's why we can decouple information for who's existing in each group at the group service with the help of route id.

There can be millions of messages getting delivered at each group, hence we need to limit the number of people at each group as it will be difficult to handle such load. (let's say 200).

We can directly send the electronic message from the user application itself, there's no work needs to be done at the gateway. Message can later be parsed using a parser & unparsed micro-service.

Facebook uses thrift for the same.

There can be group id to user id mapping present at the group service level. And that will be a one to many mapping.

And to avoid lot of duplication in the mappings we've, we can make use of consistent hashing.

It helps to reduce memory split across servers and allows to route request to the right box based on the group id. If we've requests routed on the group id, we can easily identify the users of that group.

That takes care of the routing mechanism we've.

In case the group service fails, like we send the message to the box and it fails, we can retry.

We can only retry if we know what request needs to be sent next.

One of the mechanisms for this is the message queue.

Once we give message to the message queue, it ensures that the message will be sent.

May be now, 10s later or 15s later. Those are configurable options and also how many times we are going to retry , all of these are configurable in the message queue.

If the message queue fails even after 5 retries , it can tell that it's failed and notification will be sent all the way to the client.

Group receipts will be expensive as everyone needs to view the message.

For Group / Chat Messaging we need :
1. Idempotency
2. Retrial
3. Ordering

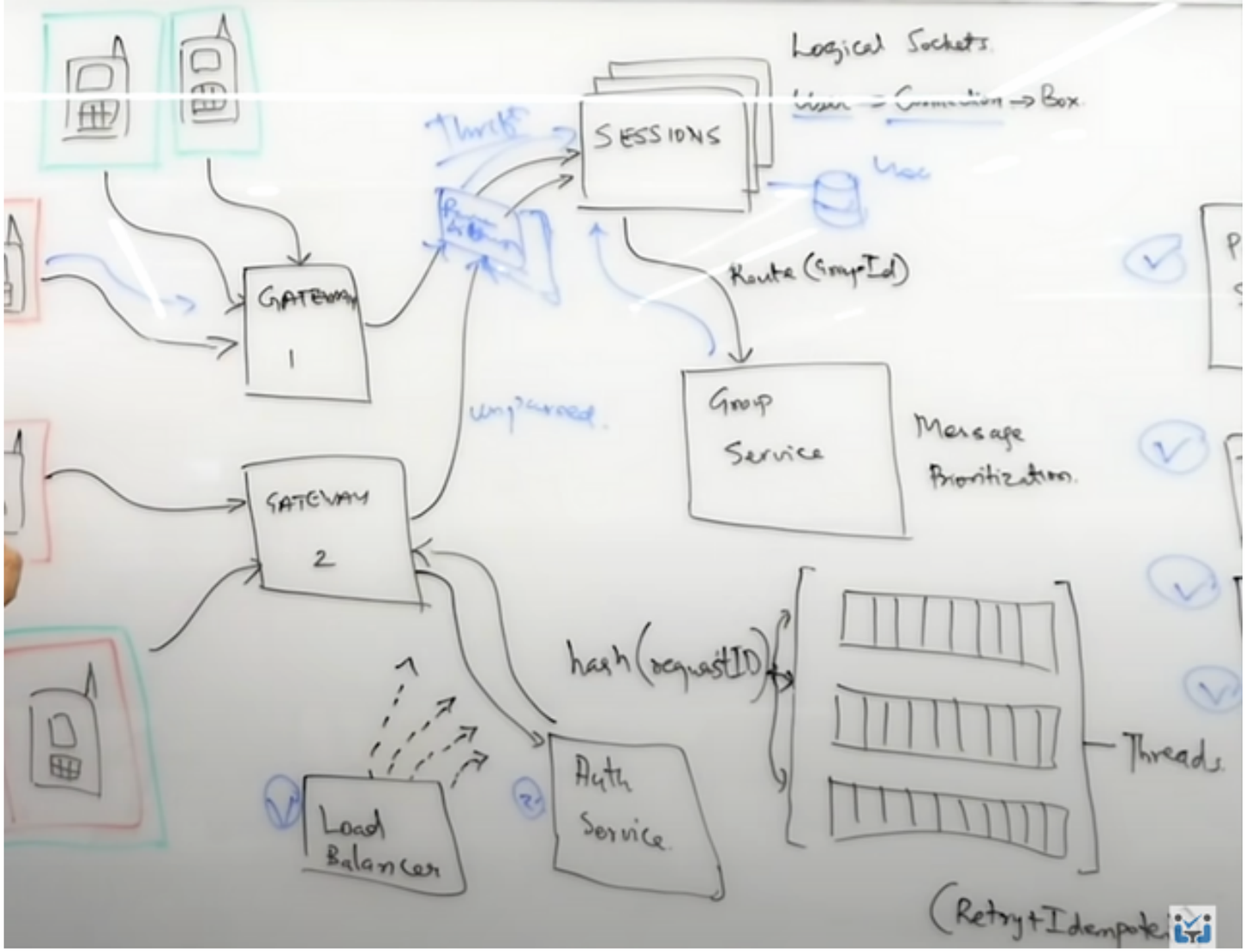
Facebook messenger de-priortizes the un-important messages incase there's a huge event to keep the system health good.

Things like last seen can be ignored during festival seasons diwali because of the load. RateLimiting principles comes into play here.

5. Image / Video Sharing

It can be understood using the tinder system design:

https://www.youtube.com/watch?v=tndzLznxq40&list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX&index=8



Useful Links:

1. https://www.youtube.com/watch?v=vvhC64hQZMk
2. https://www.enjoyalgorithms.com/blog/design-whatsapp
3. https://bytebytego.com/courses/system-design-interview/design-a-chat-system
4. https://www.theverge.com/2016/4/12/11415198/facebook-messenger-whatsapp-number-messages-vs-sms-f8-2016
5. https://www.youtube.com/watch?v=oUJbuFMyBDk&list=PLMCXHnjXnTnvo6alSjVkgxV-VH6EPyvoX&index=5 (Message Queues)

Consistent Hashing

Consistent hashing is a hashing technique that performs really well when operated in a dynamic environment where the distributed system scales up and scales down frequently.

The core concept of Consistent Hashing was introduced in the paper Consistent Hashing and RandomTrees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web but it gained popularity after the famous paper introducing DynamoDB - Dynamo: Amazon’s Highly Available Key-value Store.

Since then the consistent hashing gained traction and found a ton of use cases in designing and scaling distributed systems efficiently.

The two famous examples that exhaustively use this technique are Bit Torrent, for their peer-to-peer networks and Akamai, for their web caches. We'll dive deep into the need of Consistent Hashing, the internals of it, and more importantly along the way implement it using arrays and Binary Search.

Hash Functions

Before we jump into the core Consistent Hashing technique we first get a few things cleared up, one of which is Hash Functions. Hash Functions are any functions that map value from an arbitrarily sized domain to another fixed-sized domain, usually called the Hash Space. For example, mapping URLs to 32-bit integers or web pages' HTML content to a 256-byte string. The values generated as an output of these hash functions are typically used as keys to enable efficient lookups of the original entity.

An example of a simple hash function is a function that maps a 32-bit integer into an 8-bit integer hash space. The function could be implemented using the arithmetic operator modulo and we can achieve this by taking a modulo 256 which yields numbers in the range [0, 255] taking up 8-bits for its representation. A hash function, that maps keys to such integer domain, more often than not applies the modulo N so as to restrict the values, or the hash space, to a range [0, N-1].

A good hash function has the following properties - The function is computationally efficient and the values generated are easy for lookups - The function, for most general use cases, behaves like a pseudorandom generator that spreads data out evenly without any noticeable correlation

Now that we have seen what a hash function is, we take a look into how we could use them and build a somewhat scalable distributed system.

Building a distributed storage system

Say we are building a distributed storage system in which users can upload files and access them on demand. The service exposes the following APIs to the users

- upload to upload the file
- fetch to fetch the file and return its content

Behind the scenes the system has Storage Nodes on which the files are stored and accessed. These nodes expose the functions put_file and fetch_file that puts and gets the file content to/from the disk and sends the response to the main API server which in turn sends it back to the user.

To sustain the initial load, the system has 5 Stogare Nodes which stores the uploaded files in a distributed manner. Having multiple nodes ensures that the system, as a whole, is not overwhelmed, and the storage is distributed almost evenly across.

When the user invokes upload function with the path of the file, the system first needs to identify the storage node that will be responsible for holding the file and we do this by applying a hash function to the path and in turn getting the storage node index. Once we get the storage node, we read the content of the file and put that file on the node by invoking the put_file function of the node.

```
# storage_nodes holding instances of actual storage node objects
storage_nodes = [
    StorageNode(name='A', host='10.131.213.12'),
    StorageNode(name='B', host='10.131.217.11'),
    StorageNode(name='C', host='10.131.142.46'),
    StorageNode(name='D', host='10.131.114.17'),
    StorageNode(name='E', host='10.131.189.18'),
]


def hash_fn(key):
    """The function sums the bytes present in the `key` and then
    take a mod with 5. This hash function thus generates output
    in the range [0, 4].
    """
    return sum(bytearray(key.encode('utf-8'))) % 5


def upload(path):
    # we use the hash function to get the index of the storage node
    # that would hold the file
    index = hash_fn(path)

    # we get the StorageNode instance
    node = storage_nodes[index]

    # we put the file on the node and return
    return node.put_file(path)
```

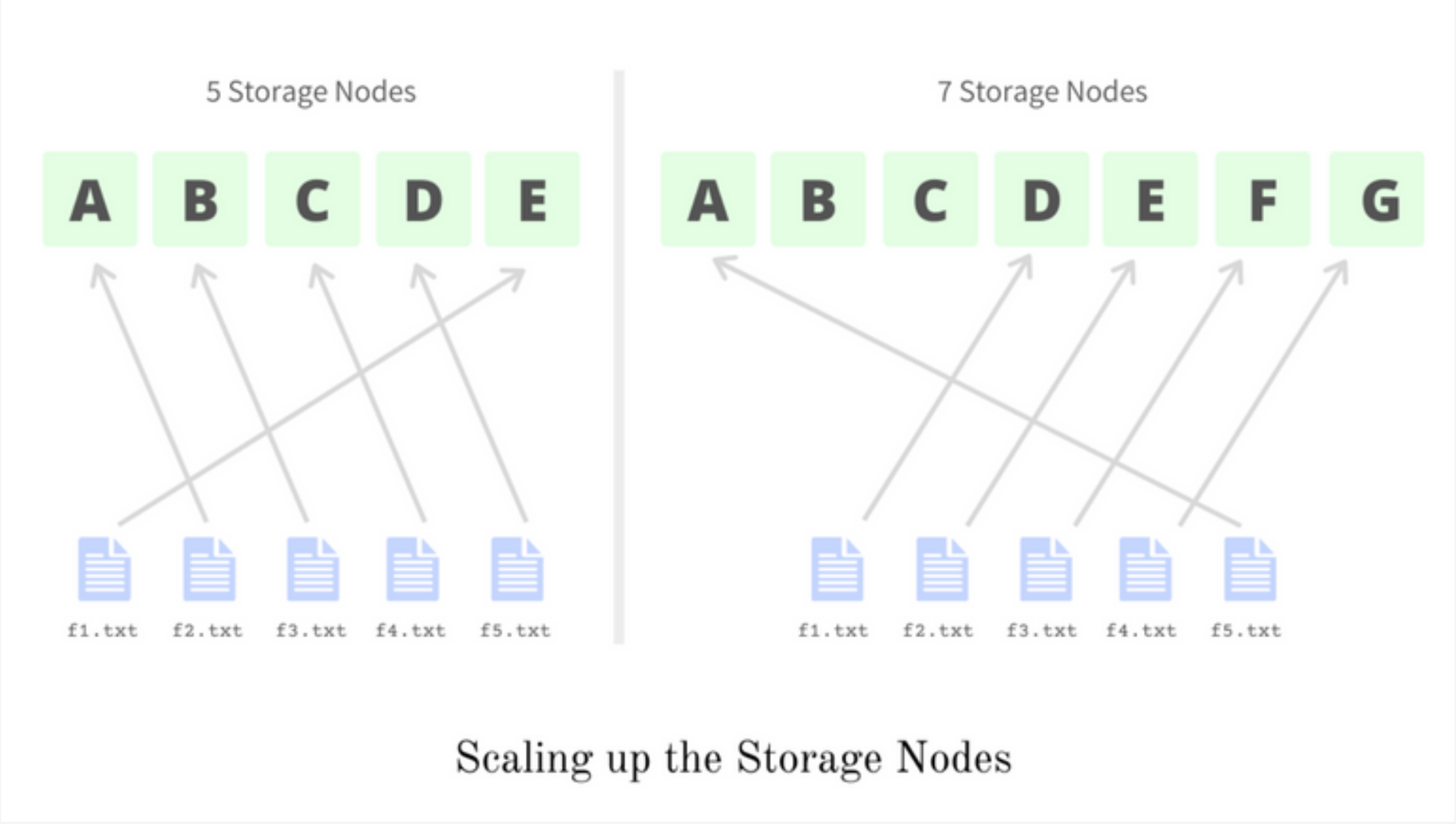
```
def fetch(path):
    # we use the hash function to get the index of the storage node
    # that would hold the file
    index = hash_fn(path)

    # we get the StorageNode instance
    node = storage_nodes[index]

    # we fetch the file from the node and return
    return node.fetch_file(path)
```

The hash function used over here simply sums the bytes and takes the modulo by 5 (since there are 5 storage nodes in the system) and thus generating the output in the hash space [0, 4]. This output value now represents the index of the storage engine that will be responsible for holding the file.

Say we have 5 files 'f1.txt', 'f2.txt', 'f3.txt', 'f4.txt', 'f5.txt' if we apply the hash function to these we find that they are stored on storage nodes E, A, B, C, and D respectively.

Things become interesting when the system gains some traction and it needs to be scaled to 7 nodes, which means now the hash function should do mod 7 instead of a mod 5. Changing the hash function implies changing the mapping and association of files with storage nodes. We first need to administer the new associations and see which files required to be moved from one node to another.

With the new hash function the same 5 files 'f1.txt', 'f2.txt', 'f3.txt', 'f4.txt', 'f5.txt' will now be associated with storage nodes D, E, F, G, A. Here we see that changing the hash function requires us to move every single one of the 5 files to a different node.



Scaling up the Storage Nodes

If we have to change the hash function every time we scale up or down and if this requires us to move not all but even half of the data, the process becomes super expensive and in longer run infeasible. So we need a way to minimize the data movement required during scale-ups or scale-downs, and this is where Consistent Hashing fits in and minimizes the required data transfer.

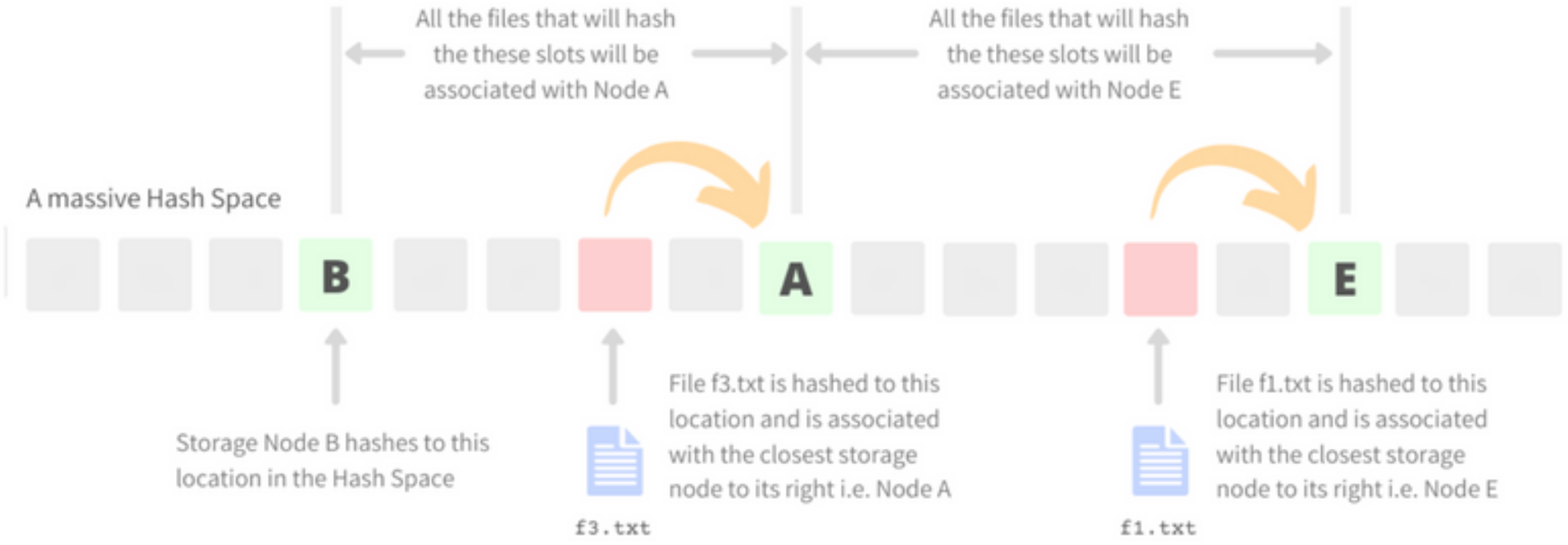
Consistent Hashing

The major pain point of the above system is that it is prone to events like scale-ups and scale-downs as it requires a lot of alterations in associations. These associations are purely driven by the underlying Hash Function and hence if we could somehow make this hash function independent of the number of the storage nodes in the system, we address this flaw.

Consistent Hashing addresses this situation by keeping the Hash Space huge and constant, somewhere in the order of [0, 2^128 - 1] and the storage node and objects both map to one of the slots in this huge Hash Space. Unlike in the traditional system where the file was associated with storage node at index where it got hashed to, in this system the chances of a collision between a file and a storage node are infinitesimally small and hence we need a different way to define this association.

Instead of using a collision-based approach we define the association as - the file will be associated with the storage node which is present to the immediate right of its hashed location. Defining association in this way helps us

- keep the hash function independent of the number of storage nodes
- keep associations relative and not driven by absolute collisions



Associations in Consistent Hashing

> Consistent Hashing on an average requires only k/n units of data to be migrated during scale up and down; where k is the total number of keys and n is the number of nodes in the system.

A very naive way to implement this is by allocating an array of size equal to the Hash Space and putting files and storage node literally in the array on the hashed location. In order to get association we iterate from the item's hashed location towards the right and find the first Storage Node. If we reach the end of the array and do not find any Storage Node we circle back to index 0 and continue the search.

The approach is very easy to implement but suffers from the following limitations

- requires huge memory to hold such a large array
- finding association by iterating every time to the right is O(hash_space)

A better way of implementing this is by using two arrays: one to hold the Storage Nodes, called nodes and another one to hold the positions of the Storage Nodes in the hash space, called keys.

There is a one-to-one correspondence between the two arrays - the Storage Node nodes[i] is present at position keys[i] in the hash space. Both the arrays are kept sorted as per the keys array.

Hash Function in Consistent Hashing

We define total_slots as the size of this entire hash space, typically of the order 2^256 and the hash function could be implemented by taking SHA-256 followed by a mod total_slots. Since the total_slots is huge and a constant the following hash function implementation is independent of the actual number of Storage Nodes present in the system and hence remains unaffected by events like scale-ups and scale-downs.

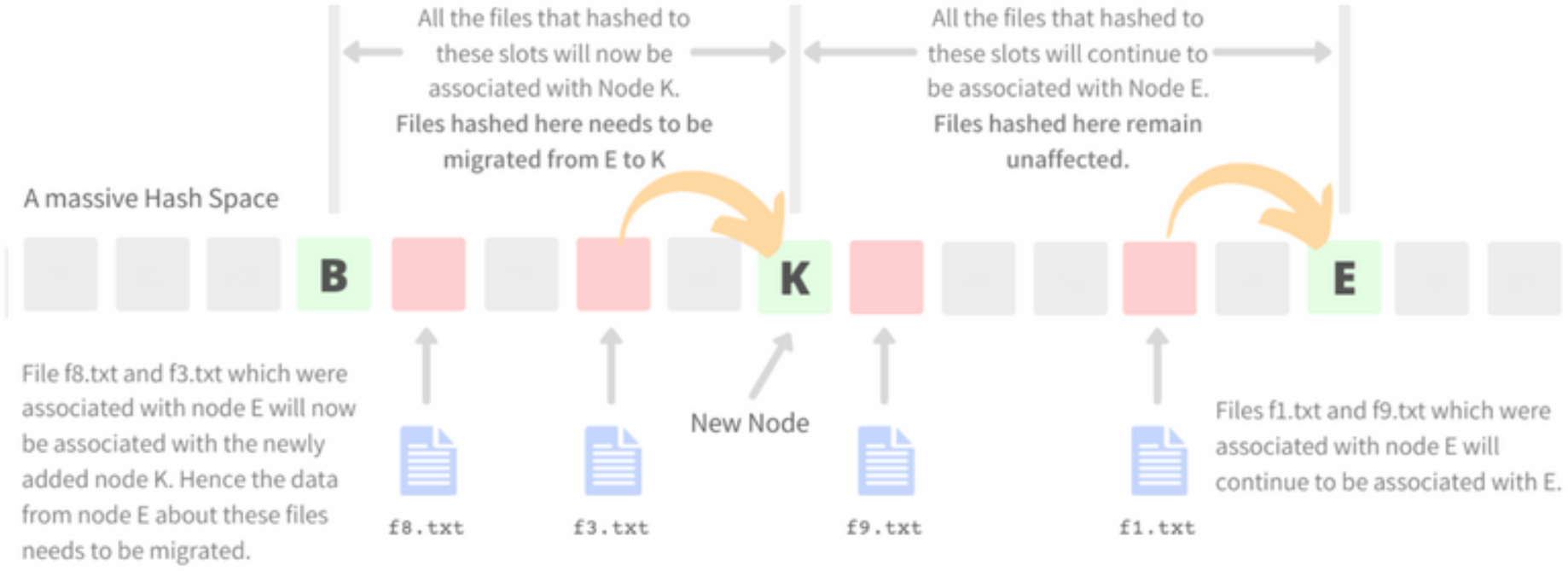
```
def hash_fn(key: str, total_slots: int) -> int:
    """hash_fn creates an integer equivalent of a SHA256 hash and
    takes a modulo with the total number of slots in hash space.
    """
    hsh = hashlib.sha256()

    # converting data into bytes and passing it to hash function
    hsh.update(bytes(key.encode('utf-8')))

    # converting the HEX digest into equivalent integer value
    return int(hsh.hexdigest(), 16) % total_slots
```

Adding a new node in the system:

When there is a need to scale up and add a new node in the system, in our case a new Storage Node, we

find the position of the node where it resides in the Hash Space
populate the new node with data it is supposed to serve
add the node in the Hash Space

When a new node is added in the system it only affects the files that hash at the location to the left and associated with the node to the right, of the position the new node will fit in. All other files and associations remain unaffected, thus minimizing the amount of data to be migrated and mapping required to be changed.



When a new node is added to the system only the files to the left of the node, associated with the node to the right of the newly added node, are affected and needs migration; associations of all other files remain intact.

Adding a new node in Consistent Hashing

From the illustration above, we see when a new node K is added between nodes B and E, we change the associations of files present in the segment B-K and assign them to node K. The data belonging to the segment B-K could be found at node E to which they were previously associated with. Thus the only files affected and that needs migration are in the segment B-K; and their association changes from node E to node K.

In order to implement this at a low-level using nodes and keys array, we first get the position of the new node in the Hash Space using the hash function. We then find the index of the smallest key greater than the position in the sorted keys array using binary search. This index will be where the key and the new Storage node will be placed in keys and nodes array respectively.

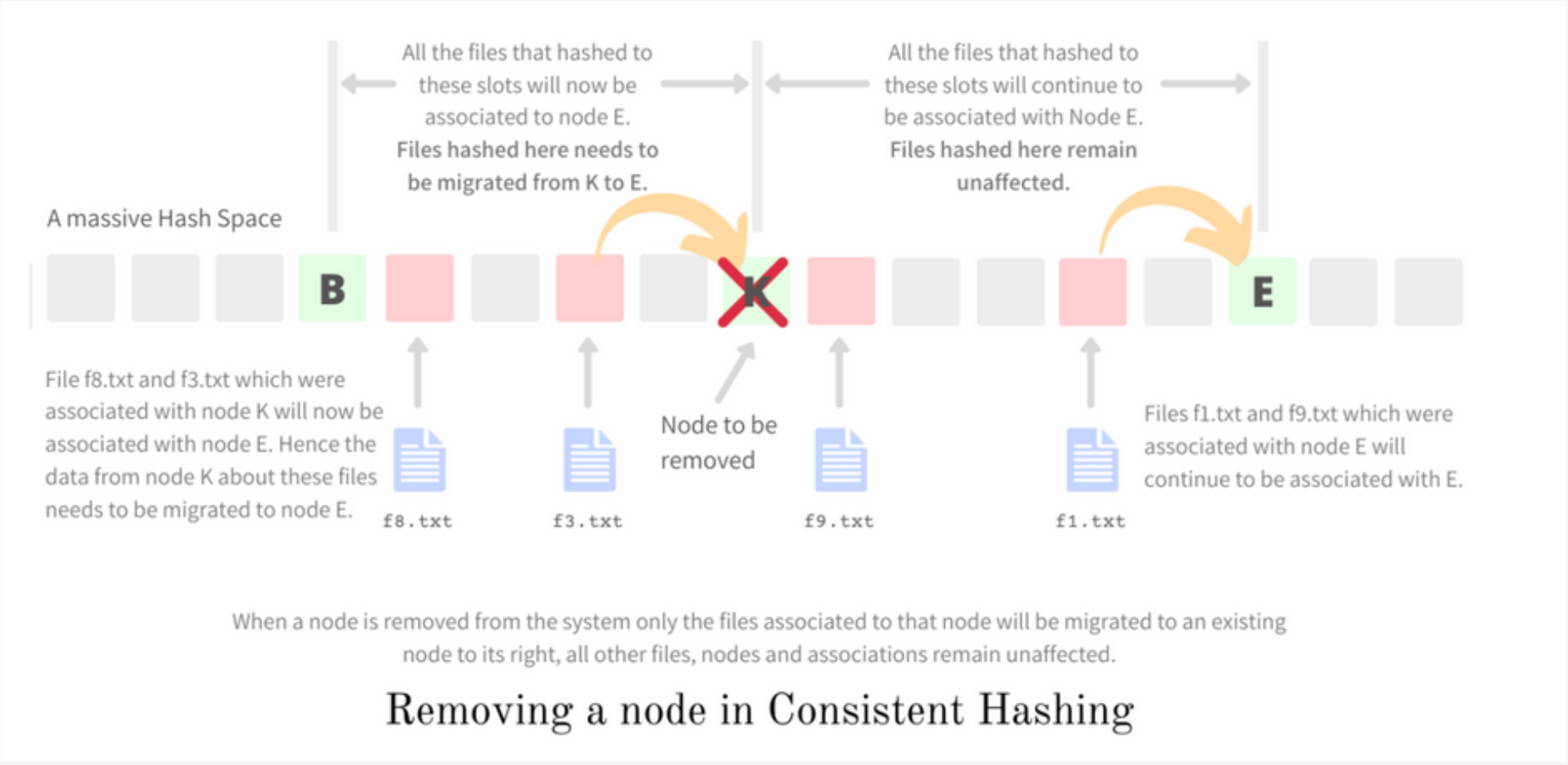
```
def add_node(self, node: StorageNode) -> int:
    """add_node function adds a new node in the system and returns the key
    from the hash space where it was placed
    """

    # handling error when hash space is full.
    if len(self._keys) == self.total_slots:
        raise Exception("hash space is full")

    key = hash_fn(node.host, self.total_slots)

    # find the index where the key should be inserted in the keys array
    # this will be the index where the Storage Node will be added in the
    # nodes array.
    index = bisect(self._keys, key)

    # if we have already seen the key i.e. node already is present
    # for the same key, we raise Collision Exception
    if index > 0 and self._keys[index - 1] == key:
        raise Exception("collision occurred")

    # Perform data migration

    # insert the node_id and the key at the same `index` location.
    # this insertion will keep nodes and keys sorted w.r.t keys.
    self.nodes.insert(index, node)
    self._keys.insert(index, key)

    return key
```

Removing a new node from the system

When there is a need to scale down and remove an existing node from the system, we

- find the position of the node to be removed from the Hash Space
- populate the node to the right with data that was associated with the node to be removed
- remove the node from the Hash Space

When a node is removed from the system it only affects the files associated with the node itself. All other files and associations remain unaffected, thus minimizing the amount of data to be migrated and mapping required to be changed.



From the illustration above, we see when node K is removed from the system, we change the associations of files associated with node K to the node that lies to its immediate right i.e. node E. Thus the only files affected and needs migration are the ones associated with node K.

In order to implement this at a low-level using nodes and keys array, we get the index where the node K lies in the keys array using binary search. Once we have the index we remove the key from the keys array and Storage Node from the nodes array present on that index.

```
def remove_node(self, node: StorageNode) -> int:
    """remove_node removes the node and returns the key
    from the hash space on which the node was placed.
    """

    # handling error when space is empty
    if len(self._keys) == 0:
        raise Exception("hash space is empty")

    key = hash_fn(node.host, self.total_slots)

    # we find the index where the key would reside in the keys
    index = bisect_left(self._keys, key)

    # if key does not exist in the array we raise Exception
    if index >= len(self._keys) or self._keys[index] != key:
        raise Exception("node does not exist")

    # Perform data migration

    # now that all sanity checks are done we popping the
    # keys and nodes at the index and thus removing the presence of the node.
    self._keys.pop(index)
    self.nodes.pop(index)

    return key
```

Associating an item to a node

Now that we have seen how consistent hashing helps in keeping data migration, during scale-ups and scale-downs, to a bare minimum; it is time we see how to efficiently we can find the "node to the right" for a given item. The operation to find the association has to be super fast and efficient as it is something that will be invoked for every single read and write that happens on the system.

To implement this at low-level we again take leverage of binary search and perform this operation in O(log(n)). We first pass the item to the hash function and fetch the position where the item is hashed in the hash space. This position is then binary searched in the keys array to obtain the index of the first key which is greater than the position (obtained from the hash function). if there are no keys greater than the position, in the keys array, we circle back and return the 0th index. The index thus obtained will be the index of the storage node in the nodes array associated with the item.

```
def assign(self, item: str) -> str:
    """Given an item, the function returns the node it is associated with.
    """
    key = hash_fn(item, self.total_slots)

    # we find the first node to the right of this key
    # if bisect_right returns index which is out of bounds then
    # we circle back to the first in the array in a circular fashion.
    index = bisect_right(self._keys, key) % len(self._keys)

    # return the node present at the index
    return self.nodes[index]
```

To implement this at low-level we again take leverage of binary search and perform this operation in O(log(n)). We first pass the item to the hash function and fetch the position where the item is hashed in the hash space. This position is then binary searched in the keys array to obtain the index of the first key which is greater than the position (obtained from the hash function). if there are no keys greater than the position, in the keys array, we circle back and return the 0th index. The index thus obtained will be the index of the storage node in the nodes array associated with the item.

References:

- https://en.wikipedia.org/wiki/Hash_function
- https://en.wikipedia.org/wiki/Consistent_hashing
- https://web.stanford.edu/class/cs168/l/l1.pdf
- https://www.akamai.com/us/en/multimedia/documents/technical-publication/consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-web-technical-publication.pdf
- https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

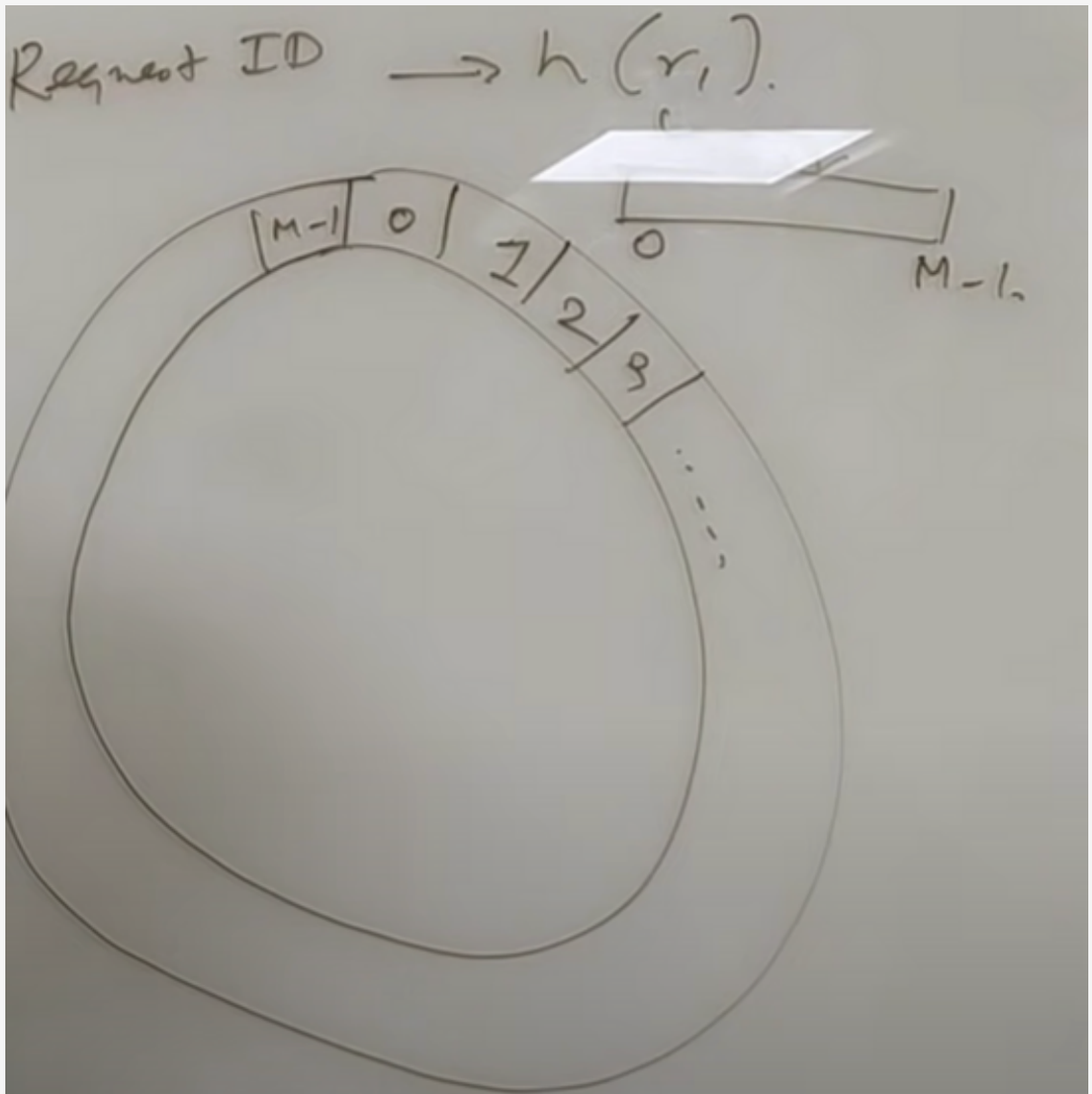
According to Gaurav Sen,

The problem with load balancing is adding and removing servers which actually changes the local data present at each server.

To avoid that we can make use of consistent hashing.

We can still hash request ids.

Instead of an array which can map the hash function from 0 to m-1. we can make use of a ring of hash.



Servers themselves have the ids. We can hash the server ids as well using the hash function and take the remainder by search space (m)

Example:
Server 1 position = h(0) % 30 = 49 % 30 = 19

Various servers will be hashed this way.

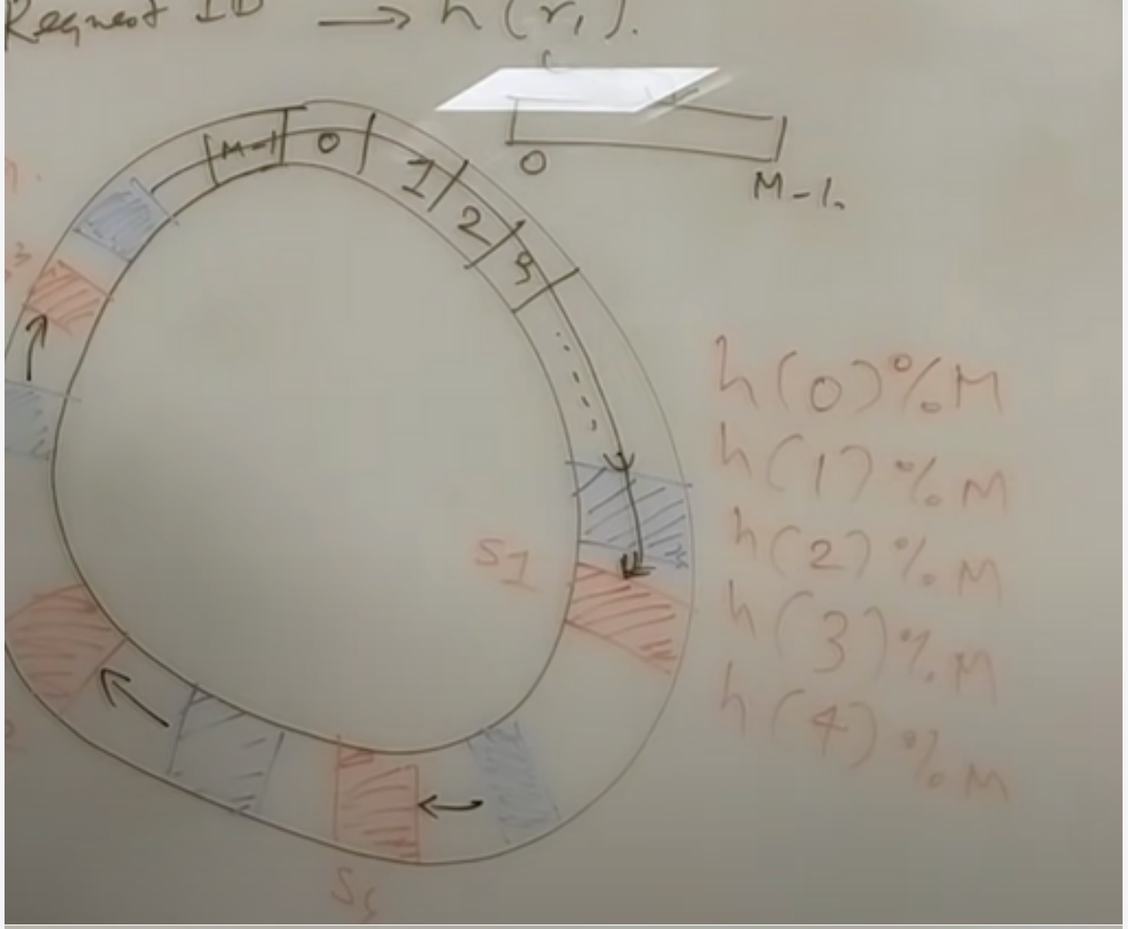
Whenever request comes in to the ring, we'll go clockwise and find the nearest server.

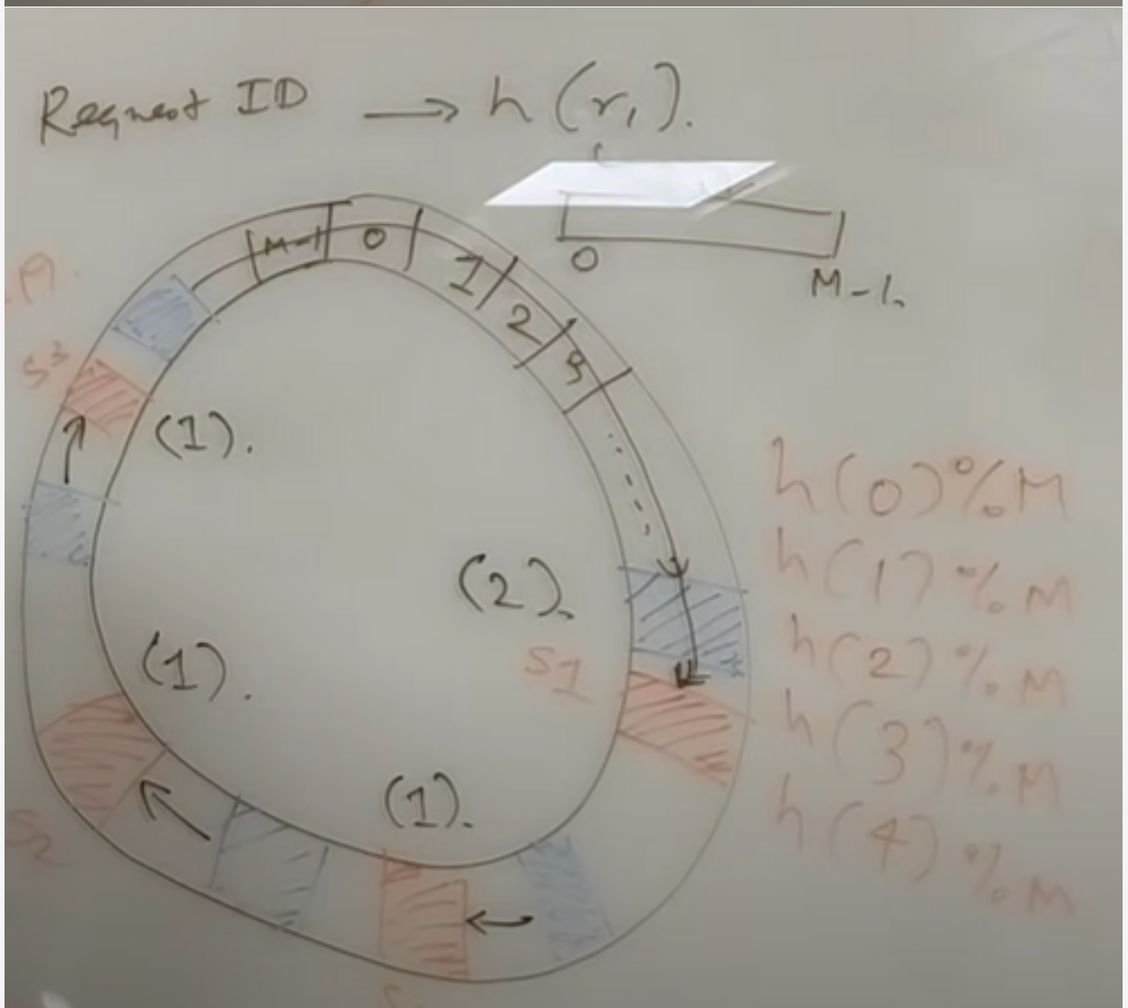
That server will be serving that request.

Hence, one server can serve multiple requests depending upon the circular position.

For example, S1 can have a load of 2 requests.

Since the hashes are uniform, requests will be uniformed and the load as well.


Request ID → h(r1).
0
M-1.
h(0)%M
h(1)%M
h(2)%M
h(3)%M
h(4)%M


Request ID → h(r1).
0
M-1.
(1).
(2).
(1).
(1).
h(0)%M
h(1)%M
h(2)%M
h(3)%M
h(4)%M

Load Factor on an average turns out to be 1/N on average (expected).

If we've a new server, any req coming there will be served by that server. Earlier that was served by the next nearest server.

Change in the servers load will be much lesser than what was there previously.

Lets say s1 goes down , next server will serve the s1's requests. All of the servers will be served by s4.

We'll have half of the load on a single server.

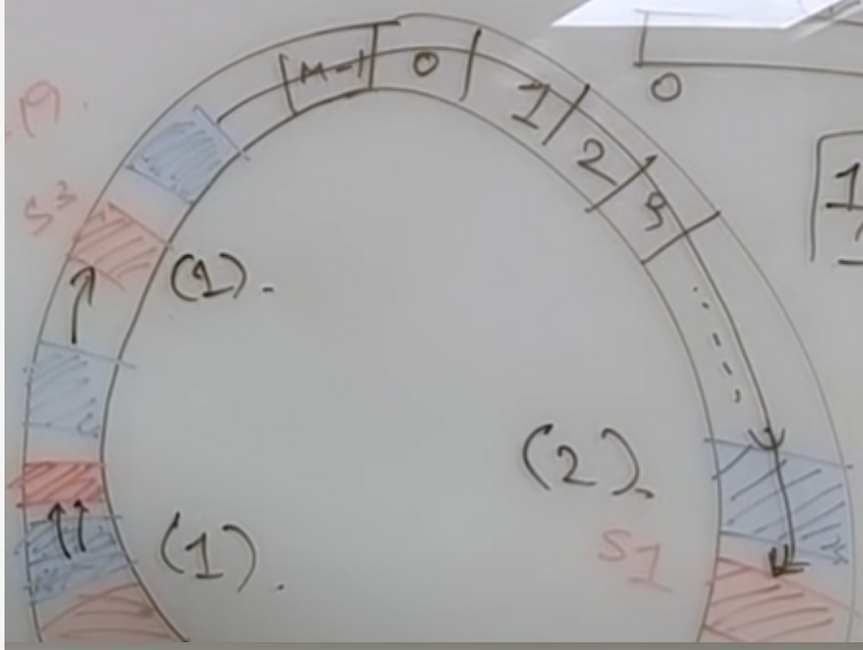
We can have multiple hash functions to solve this.

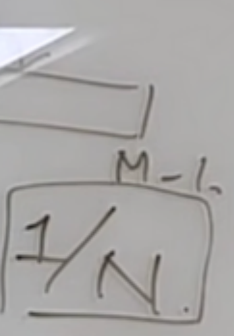
If we choose the k effectively, we can entirely remove the load on one of the service.

As load is expected to be distributed uniformly.

Consistent Hashing is used by web caches, databases, and effective load balancing.

Request ID $\longrightarrow h(r_1)$.

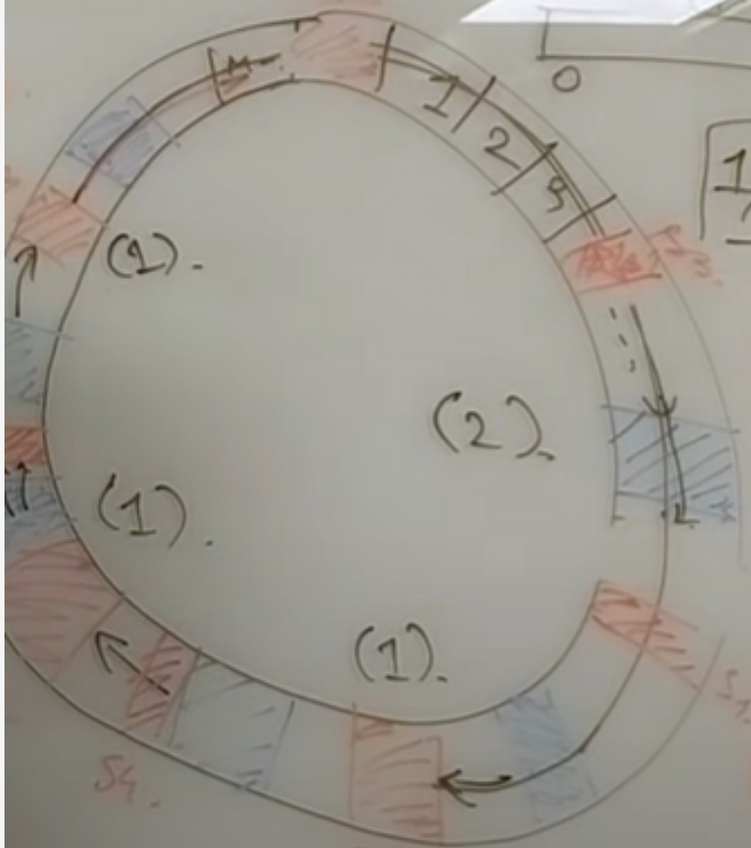


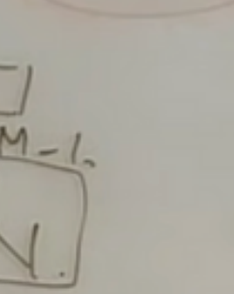


$h(0)\%M$
$h(1)\%M$
$h(2)\%M$

K hash functions.

Request ID $\longrightarrow h(r_1)$.





$h_1(0)\%M$ $h_2 \rightarrow$
$h_1(1)\%M$ $h_2 \rightarrow$
$h_1(2)\%M$ $h_2 \rightarrow$
$h_1(3)\%M$ $h_2 \rightarrow$
$h_1(4)\%M$ $h_2 \rightarrow$
$h_1(5)\%M$

Content Delivery Networks (CDNs)

Content distribution networks (or content delivery networks) is a network of proxy servers and data centers that locate them. The main function of CDNs is to ensure that server performance is maintained at scale. They are extremely crucial in reducing latency while fetching static assets from a particular server.

A content distribution network can have a huge network of servers that often spreads out globally across data centers. CDNs reduce latency and load times by keeping users close to a CDN, regardless of how far away they are from the actual server. With content distribution networks, static assets which mostly include videos, photos, and web pages coded in HTML/CSS/Javascript, aren’t fetched directly. Instead, users can access a cached copy of the assets. This massively reduces request latency and boosts server performance and also saves huge amounts of network bandwidth.

Push and Pull CDNs

Push and pull CDNs are the two main CDN functions to populate a CDN cache. In a push CDN, engineers will have to push files to the CDN when a user makes a request. A pull CDN network will automatically fetch the static asset from the origin server in case the file isn’t present when the user makes a request.

In a push CDN, if the origin server sends an asset, the CDN saves it in its cache and doesn't fetch the asset again when the user places a request.

Both push and pull CDNs come with their own set of advantages and disadvantages.

In a push CDN, developers have to constantly push files to make sure the assets are up to date. If it isn't updated, the user won't be able to fetch the asset when he/she makes a request. With pull CDNs, engineers don't have to constantly push files, thereby requiring lesser maintenance.

Pull CDNs are more popular these days than push CDNs. That's primarily because pull CDNs are much easier to maintain given the way they operate and function. There is a disadvantage associated with pull CDNs - assets become stale after a point and users cannot fetch the updated version of the asset if the cache isn't updated. Usually, pull CDNs attach a timestamp with the asset and cache an asset for 24 hours (default period). Pull CDNs also support cache-control headers and offer more flexibility concerning caching assets.

Popular Content Delivery Networks:

Most cloud computing companies have their own CDNs, mostly because they can integrate these CDNs with their service offerings. Below are some popular CDNs that are operated by cloud and computing companies including Amazon, Oracle, and Microsoft, among others:

- Azure CDN
- AWS Cloudfront
- Cloudflare CDN

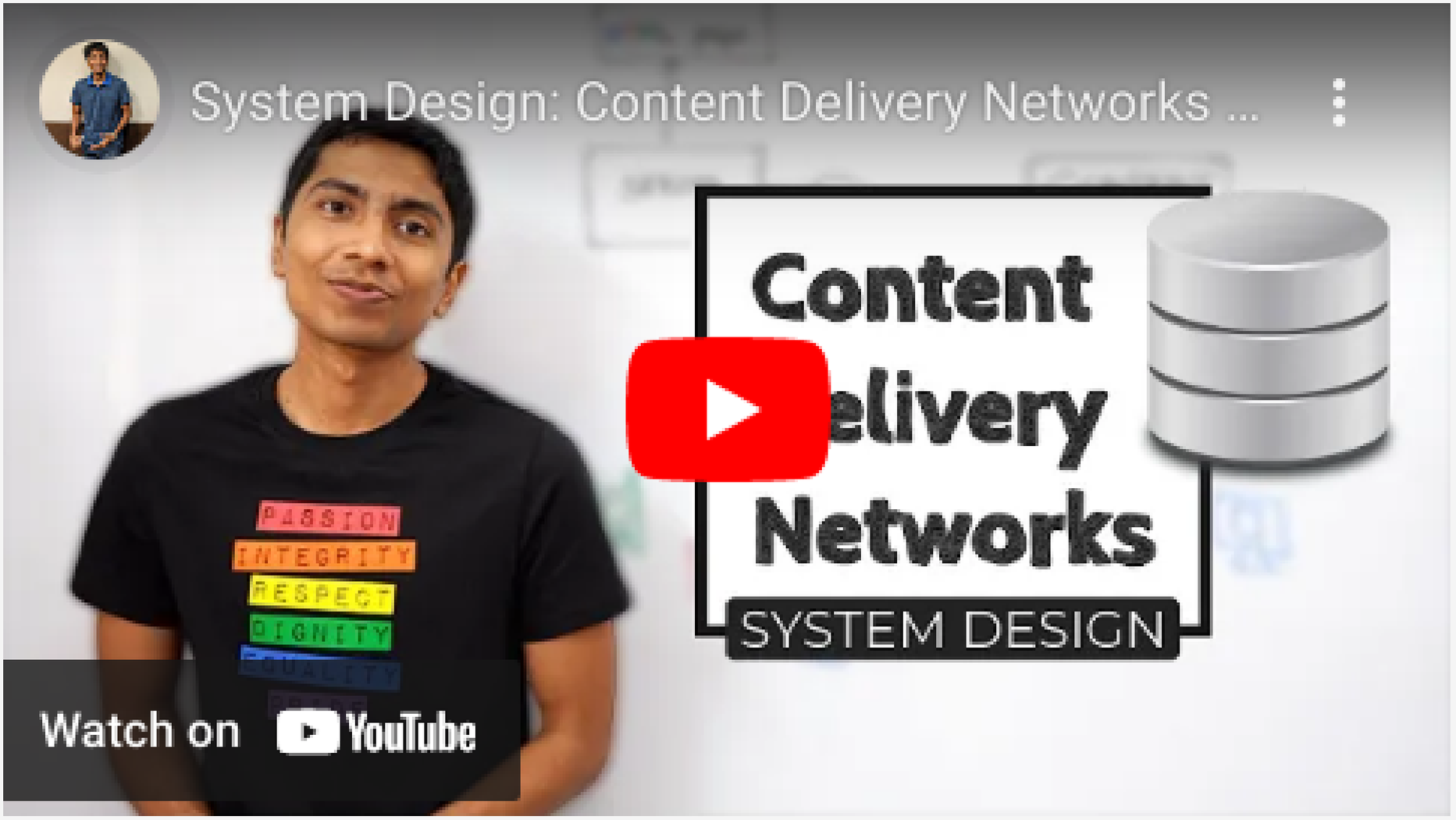
- Oracle CDN
- GCP Cloud CDN

Content Distribution Networks Interview Questions for Systems Design Interviews

For systems design interviews, content delivery systems is a popular topic around which questions are commonly asked. Being adept with the popular types of CDNs, their architecture, and how they work is essential if you wish to ace design interviews at top firms.

Below are some sample questions on CDNs for systems design interviews

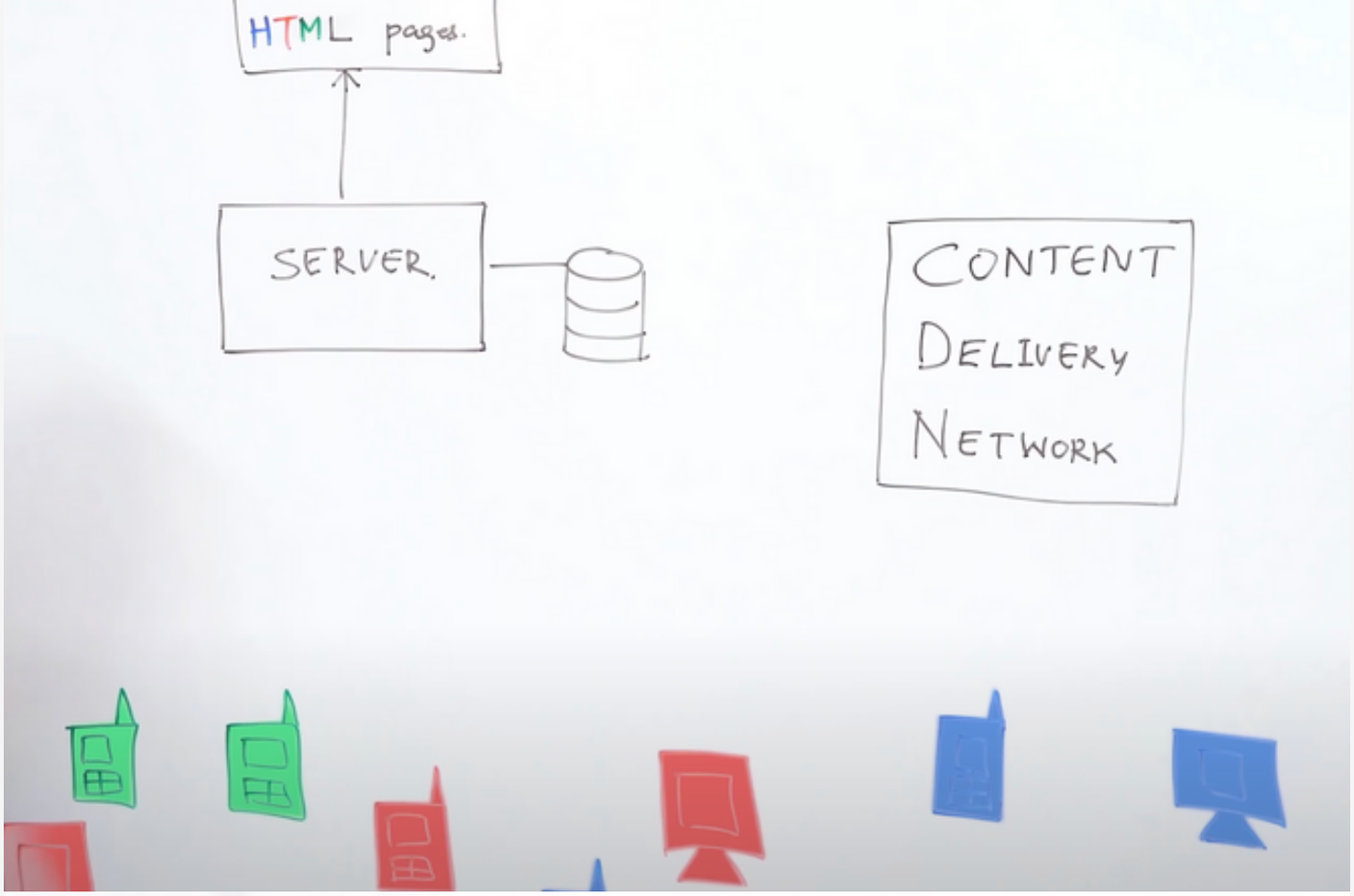
- Explain how Google uses CDNs in its services
- How do CDNs work on gaming platforms?
- Explain how caching takes place in CDNs with an example
- What are some advantages of using static Javascript files?
- Explain the difference between push and pull CDNs
- What are CDN edge servers?
- When is it ideal to use cache busting?
- What is a CDN origin server?

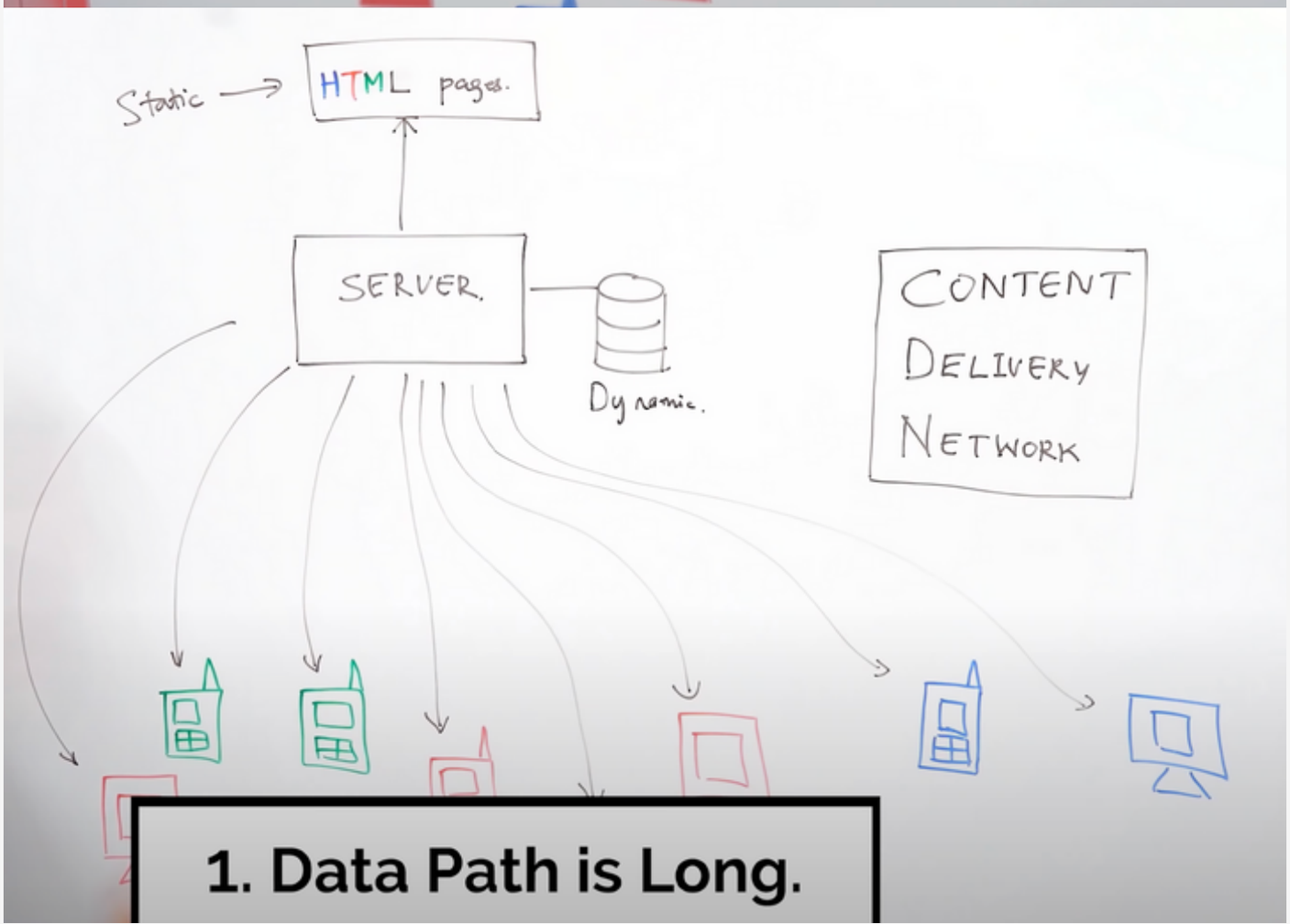


According to Gaurav Sen, whenever the user needs to connect to the server, client hits the server and asks for the HTML page.

This will cause the following issues:
1. Data Path is Long.
2. File type depends on user device.

In order to fix this, we want to have:
1. Caching
2. Customized Data (Device, Location)
3. Fast web page serving


HTML pages.
SERVER.
CONTENT
DELIVERY
NETWORK


Static
HTML pages.
SERVER.
Dynamic.
CONTENT
DELIVERY
NETWORK
1. Data Path is Long.

Before the server, we can have a another entity called global cache. The cache is a single point of failure.

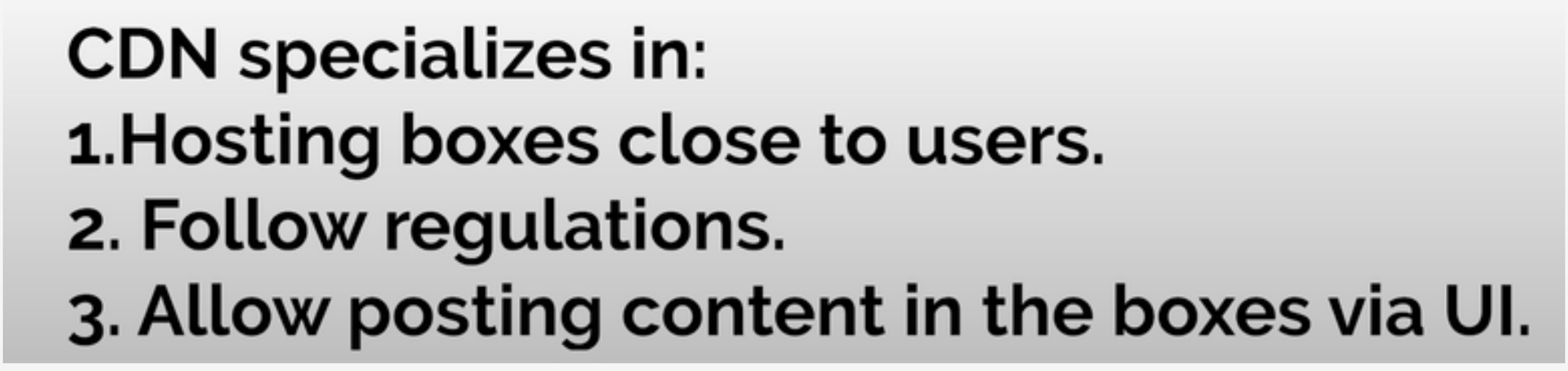
If that crashes, everything crashes. To solve this we can have a distributed cache.

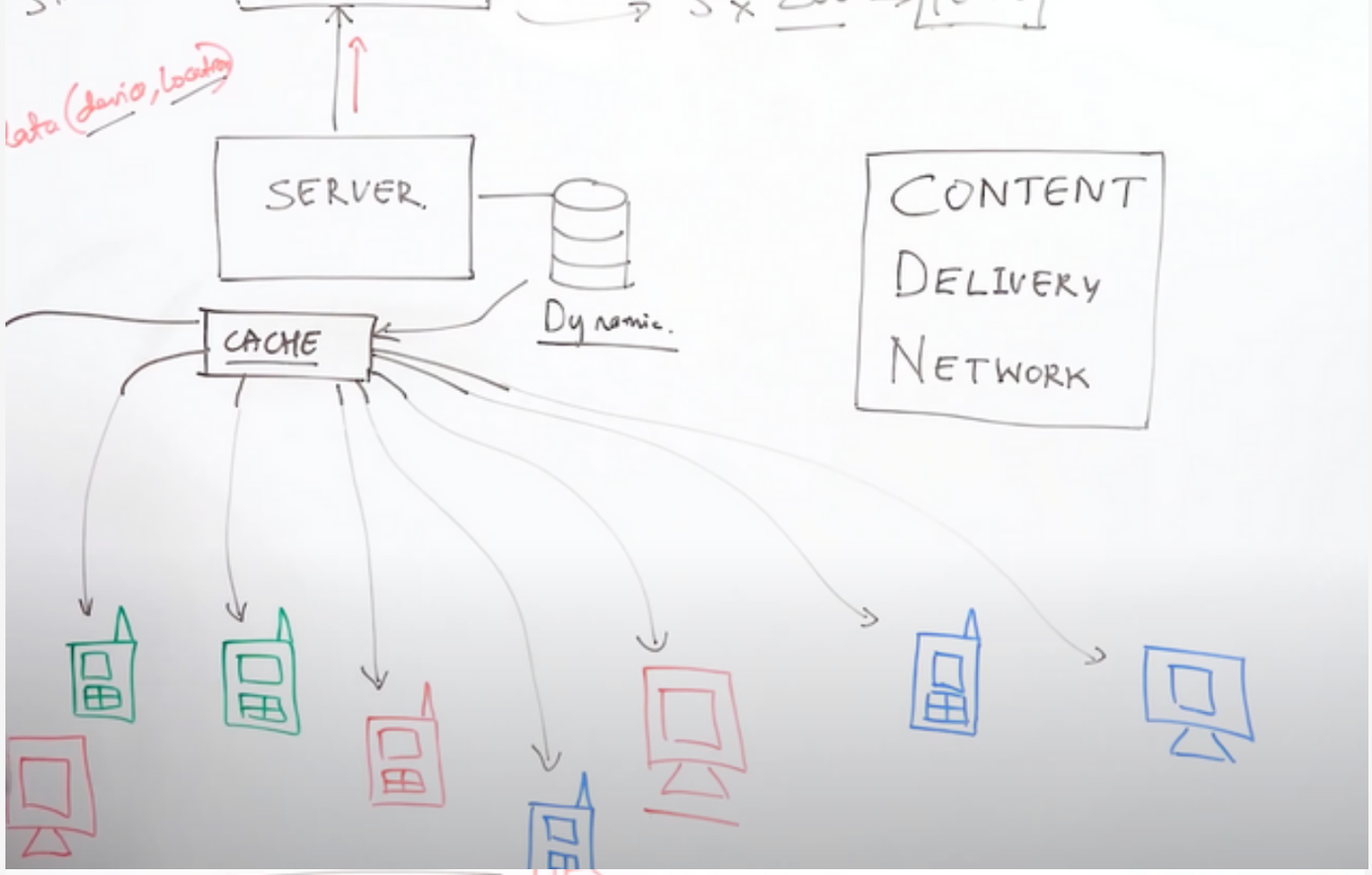
And based on the location, we can shard horizontally. In this way we can reduce the latency.

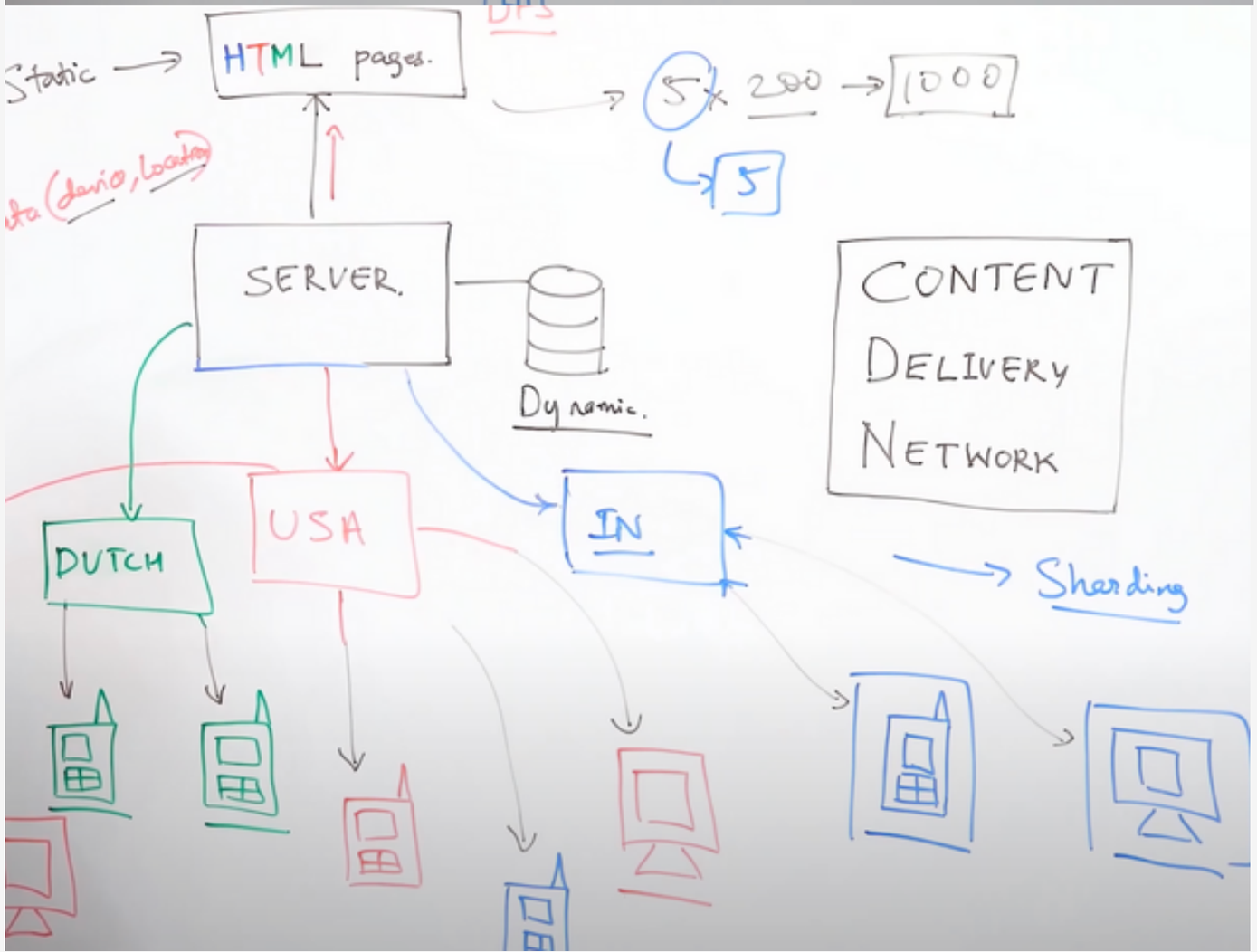
In a distributed system, we should have access to the source of truth.

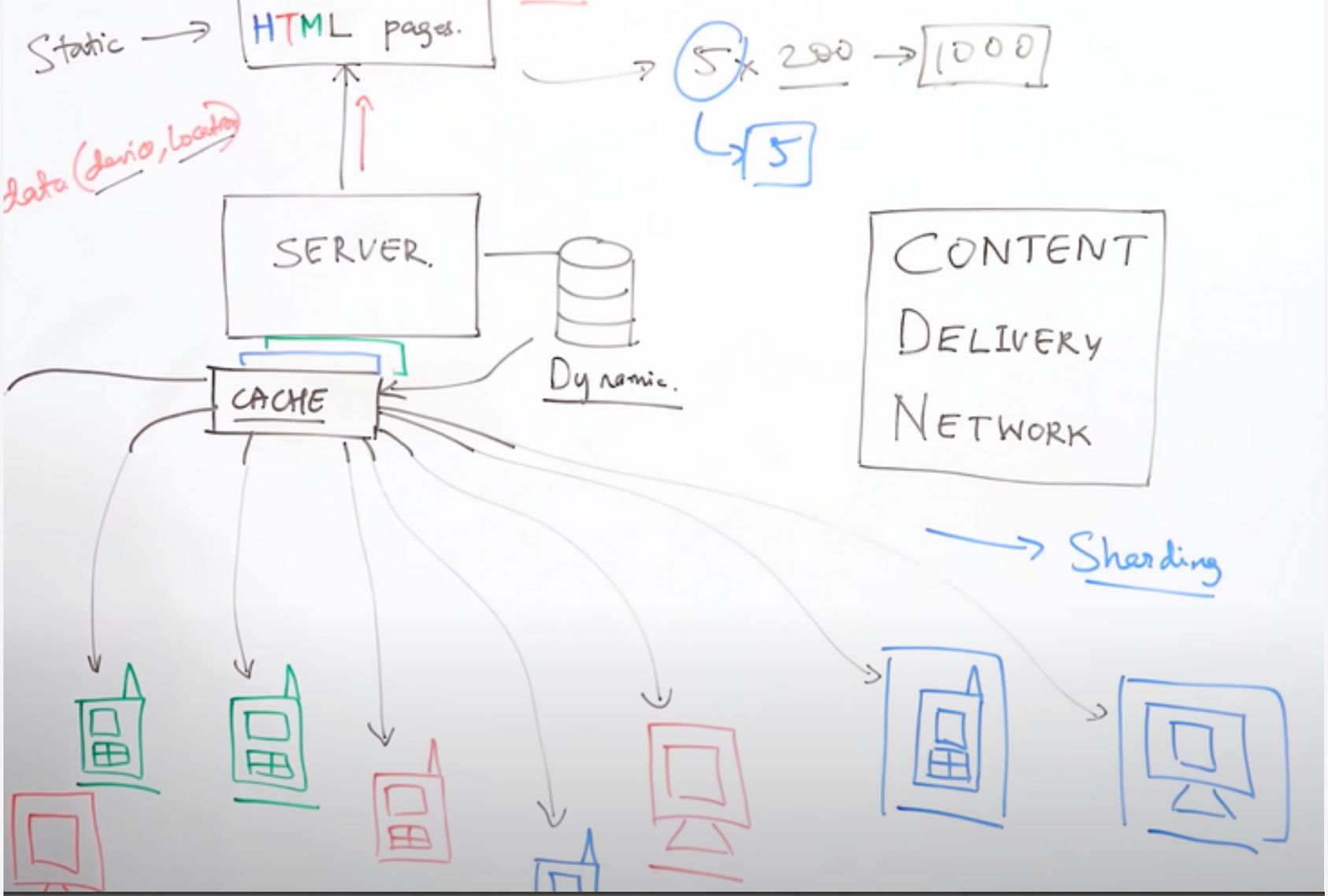
It will be difficult to make sure that the cache has following properties:
1. Available in different countries
2. Follows regulations
3. Serves the latest content

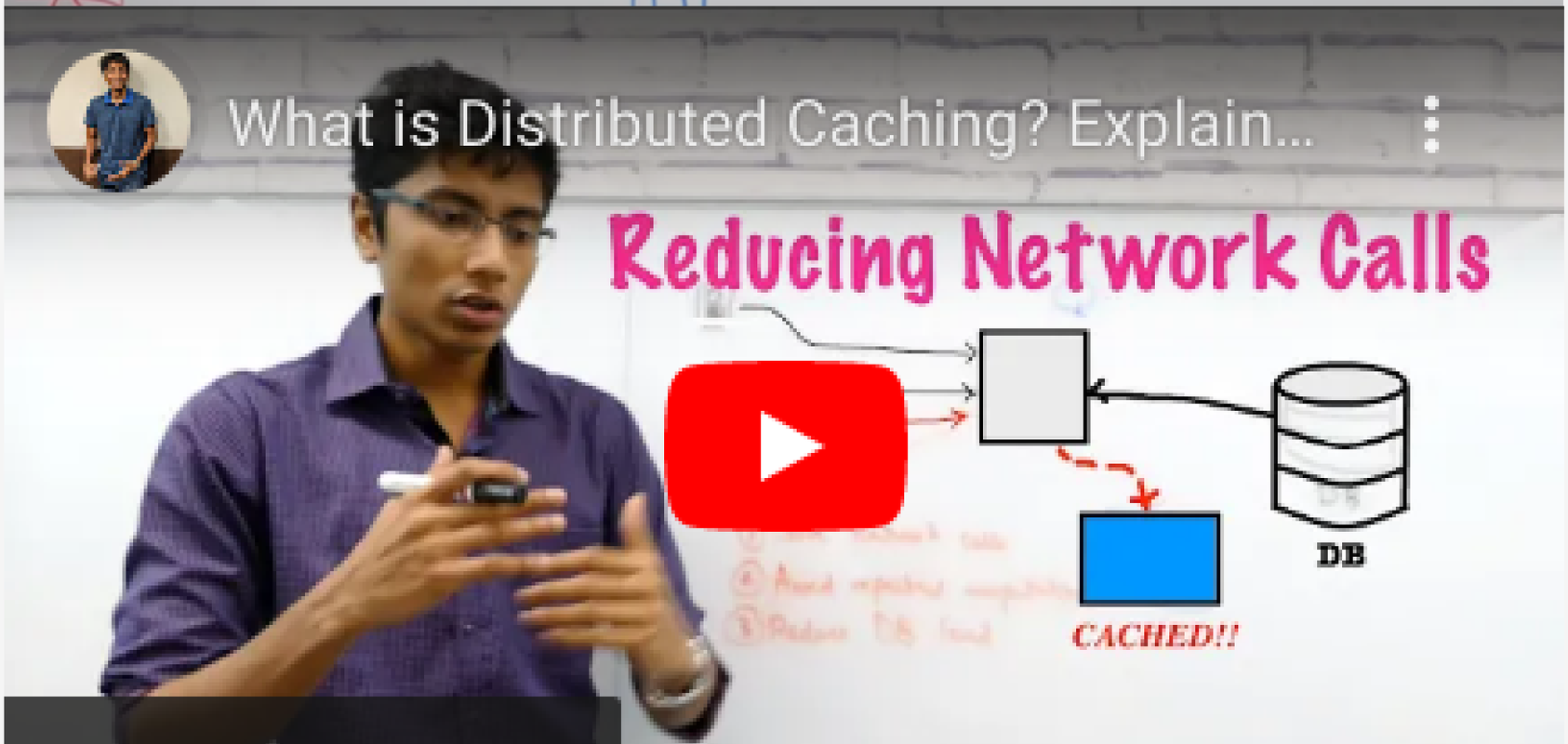
CDN takes care of all that stuff. Most popular CDN is the AWS CloudFront



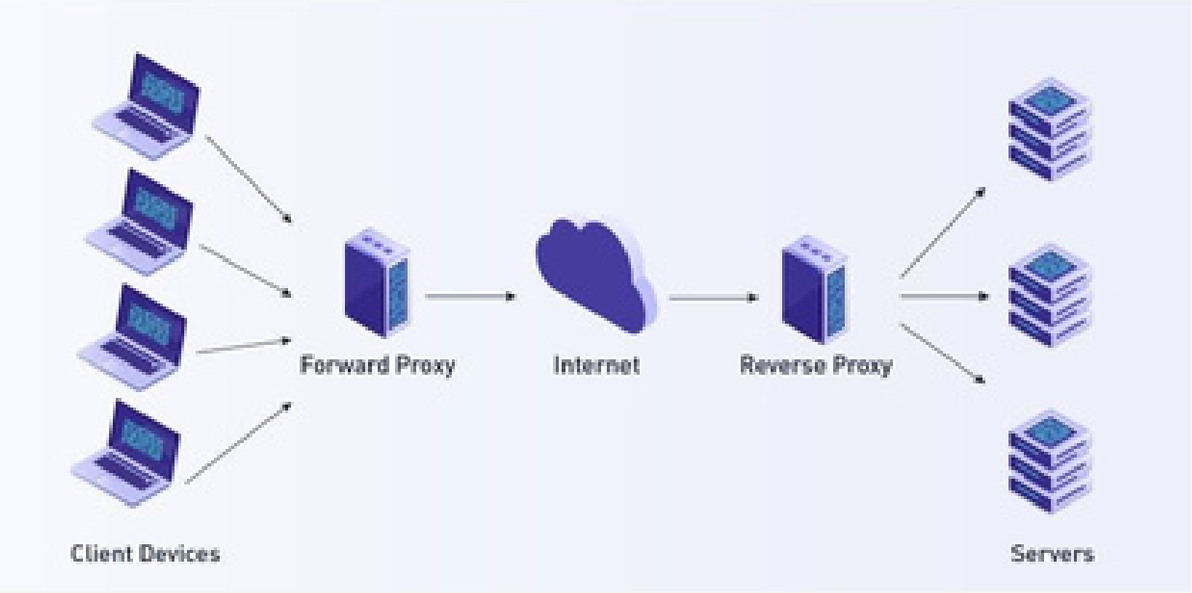

SERVER
Dynamic
CACHE
CONTENT
DELIVERY
NETWORK


Static
HTML pages.
5 x 200 → 1000
5
SERVER
Dynamic
CONTENT
DELIVERY
NETWORK
DUTCH
USA
IN
Sharding


Static
HTML pages.
5 x 200 → 1000
5
Data (data, location)
SERVER
CACHE
Dynamic.
CONTENT
DELIVERY
NETWORK
Sharding


What is Distributed Caching? Explain...
Reducing Network Calls
DB
CACHED!!

Proxy Server



Proxy Server: A proxy server is an intermediate server between a client and the destinations that they browse. It's one of the primers of system design. Following are the types of proxy servers:

- Open Proxy Server
- Forward Proxy Server
- Reverse Proxy Server

Open Proxy Server:

An open proxy is a type of proxy server that is accessible by any Internet user. Generally, a proxy server only allows users within a network group (i.e. a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service.

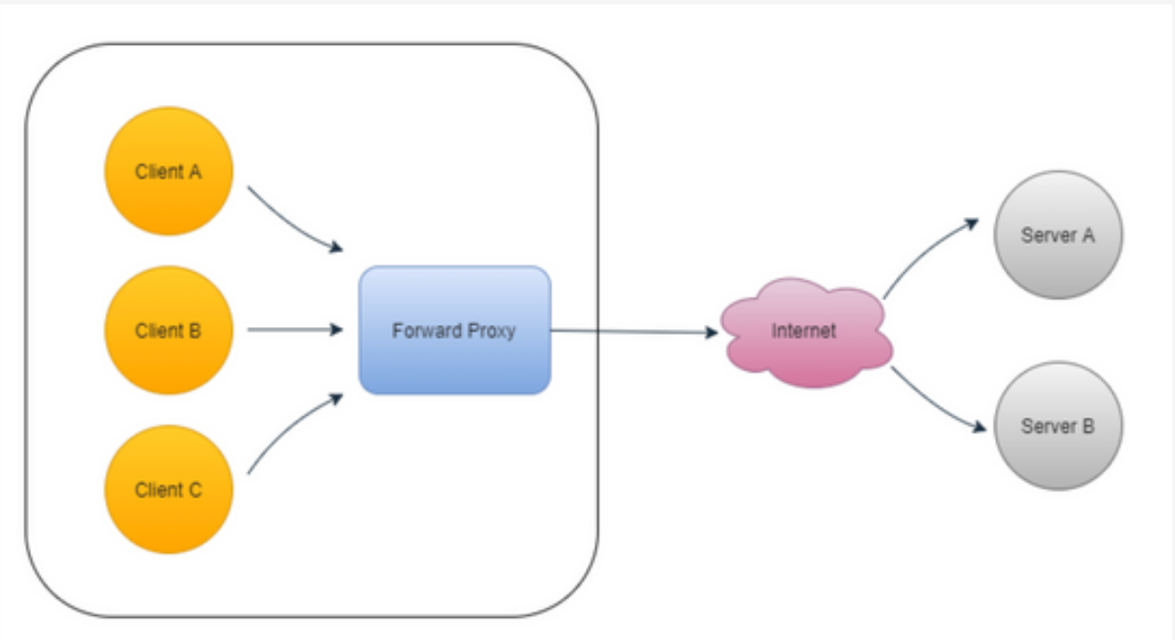
Forward Proxy Server: It is a proxy server that forwards a web request on behalf of the client. The server resides on the client-side.

How does it work?

When a client requests web content it goes to the proxy server first because the client only knows the proxy server's IP. Then proxy server forwards it to the webServer. When the proxy server gets the response from the web server, it sends back the response to the client. The actual Client is unknown to the web server in this scenario.

Benefits of forward proxy server:

- Hiding identity
- Bypassing blocked websites
- Caching
- Improves security
- Content Filtering



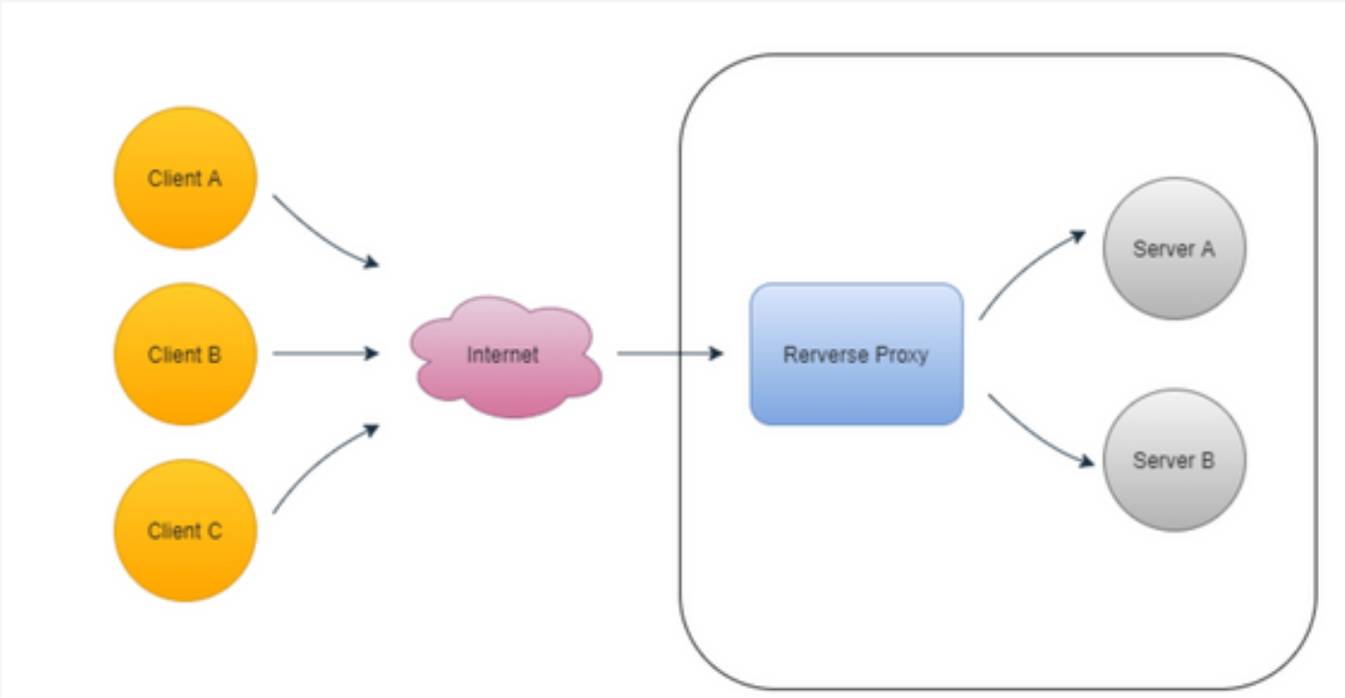
Reverse Proxy Server: On the other hand, a reverse proxy server is a proxy server that distributes the web request to the server based on request headers. The server resides on the server-side.

How does it work?

In this case, when a client sends a request to a server if that server is implemented reverse proxy, then the request first goes to the reverse proxy server without the client knowing it. Hence the client thinks it is directly interacting with the server, and for the client, there are no other servers here, and it thinks that the one it is interacting with is the server it wanted to send a request to.

Advantages of reverse proxy server:

- Security
- Load Balancing
- Caching
- Compression



In a word, forward proxy (server) acts on behalf of the client and ensures the client's anonymity, on the other hand, reverse proxy (server) acts on behalf of the server and hides the actual server from the client.

Load Balancer: A load balancer distributes incoming client requests among a group of servers, in each case returning the response from the selected server to the appropriate client.

So, it’s similar to reverse proxy as both types of applications sit between clients and servers, accepting requests from the former and delivering responses from the latter. What are the differences between reverse proxy and load balancer?

Reverse Proxy vs Load Balancer:

Load Balancer:

- Specific task - a server or device that balances inbound requests across two or more web servers to spread the load
- Can deal with the various protocols (http, dns, ftp, smtp etc.)

Reverse Proxy (server):

- A reverse proxy (server) is specific to web(http) servers.
- Load balancing capability
- Caching capability
- Compression
- SSL Termination
- Security