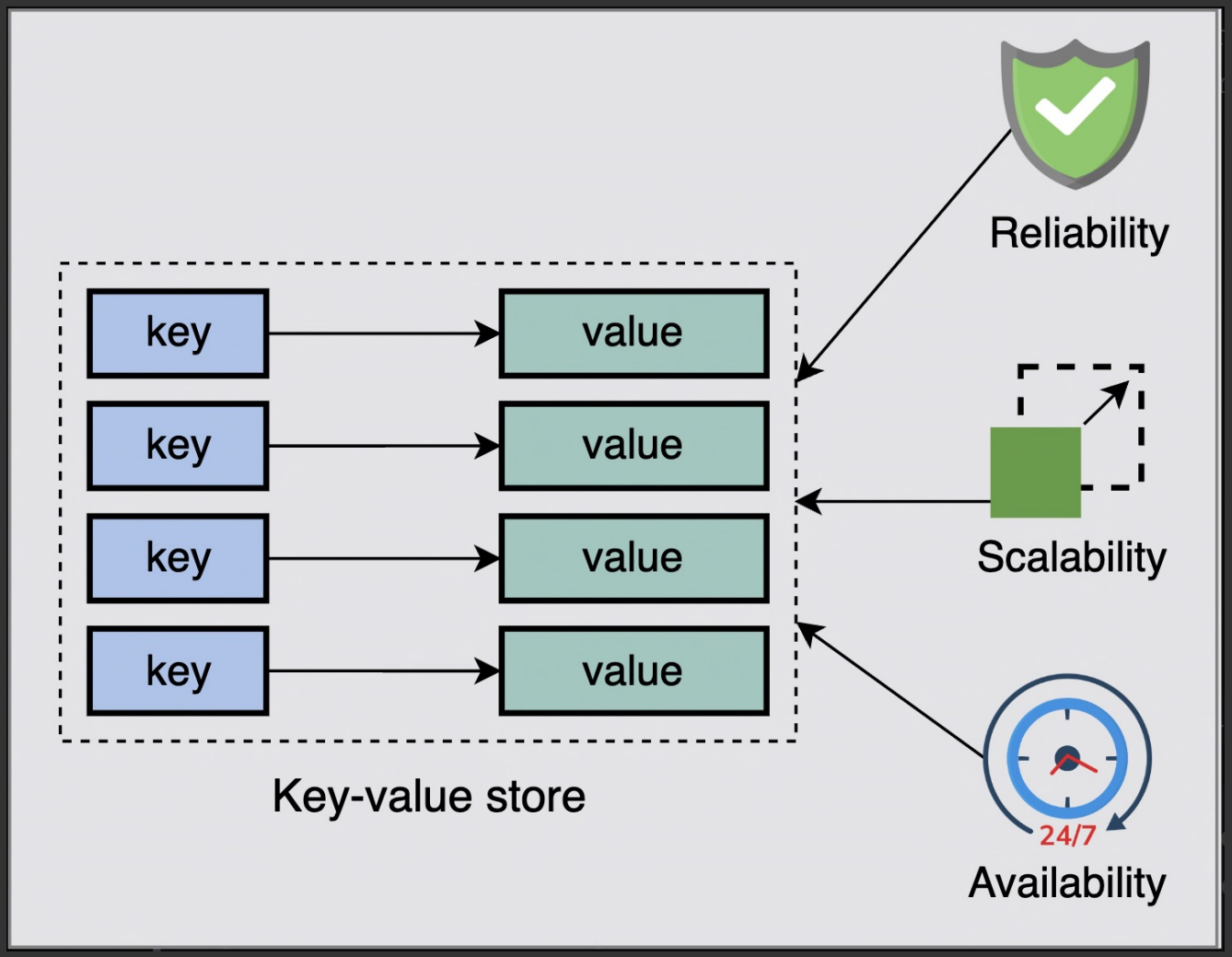
Key value store are distributed hash tables (DHTs)

In a key-value store, a key binds to a specific value and doesn't assume anything about the structure of the value. A value can be a blob, image, server name, or anything the user wants to store against a unique key.



Design of Key-Value Store

→ Typically → Key-Value stores are expected to offer functions such as get & put.

Functional Requirements

Configurable service: Some applications might have a tendency to trade strong consistency for higher availability. We need to provide a configurable service so that different applications could use a range of consistency models. We need tight control over the trade-offs between availability, consistency, cost-effectiveness, and performance

Ability to always write (when we picked “A” over “C” in the context of CAP): The applications should always have the ability to write into the key-value storage. If the user wants strong consistency, this requirement might not always be fulfilled due to the implications of the CAP theorem.

Hardware heterogeneity: We want to add new servers with different and higher capacities, seamlessly, to our cluster without changing or upgrading existing servers. Our system should be able to accommodate and leverage different capacity servers, ensuring correct core functionality (get and put data) while balancing the workload distribution according to each server’s capacity. This calls for a peer-to-peer design with no distinguished nodes.

Non-Functional Requirement

Scalability: Key-value stores should run on tens of thousands of servers distributed across the globe. Incremental scalability is highly desirable. We should add or remove the servers as needed with minimal to no disruption to the service availability. Moreover, our system should be able to handle an enormous number of users of the key-value store.

Fault tolerance: The key-value store should operate uninterrupted despite failures in servers or their components.

Why do we need to run key-value stores on multiple servers?

Hide Answer
A single-node-based hash table can fall short due to one or more of the following reasons:

No matter how big a server we get, this server can't meet data storage and query requirements.

Failure of this one mega-server will result in service downtime for everyone.

So, key-value stores should use many servers to store and retrieve data.

Assumptions

- The Data Centers hosting the service are trusted
- All the required authentication & authorization are already completed
- User requests & responses are relayed over HTTPS.

The API call to get a value should look like this:

```
get(key)
```

Parameter.	Description
key.	It’s the key against which we want to get value.

The API call to put the value into the system should look like this:

```
put(key, value)
```

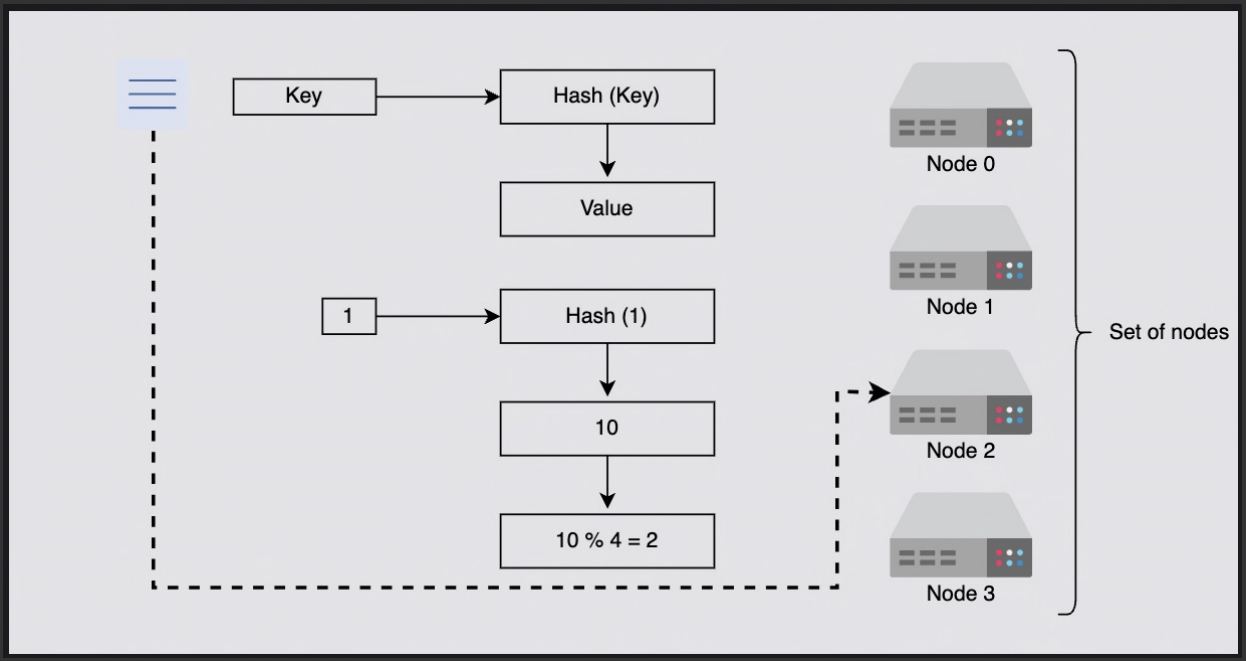
Parameter	Description
`key`	It's the `key` against which we have to store `value`.
`value`	It's the object to be stored against the `key`.

We often keep hashes of the value (and at times, value + associated key) as metadata for data integrity checks. Should such a hash be taken after any data compression or encryption, or should it be taken before?

The correct answer might depend on the specific application. Still, we can use hashes either before or after any compression or encryption. But we’ll need to do that consistently for put and get operations.

Data type
The key is often a primary key in a key-value store, while the value can be any arbitrary binary data.

Ensure Scalability & Replication

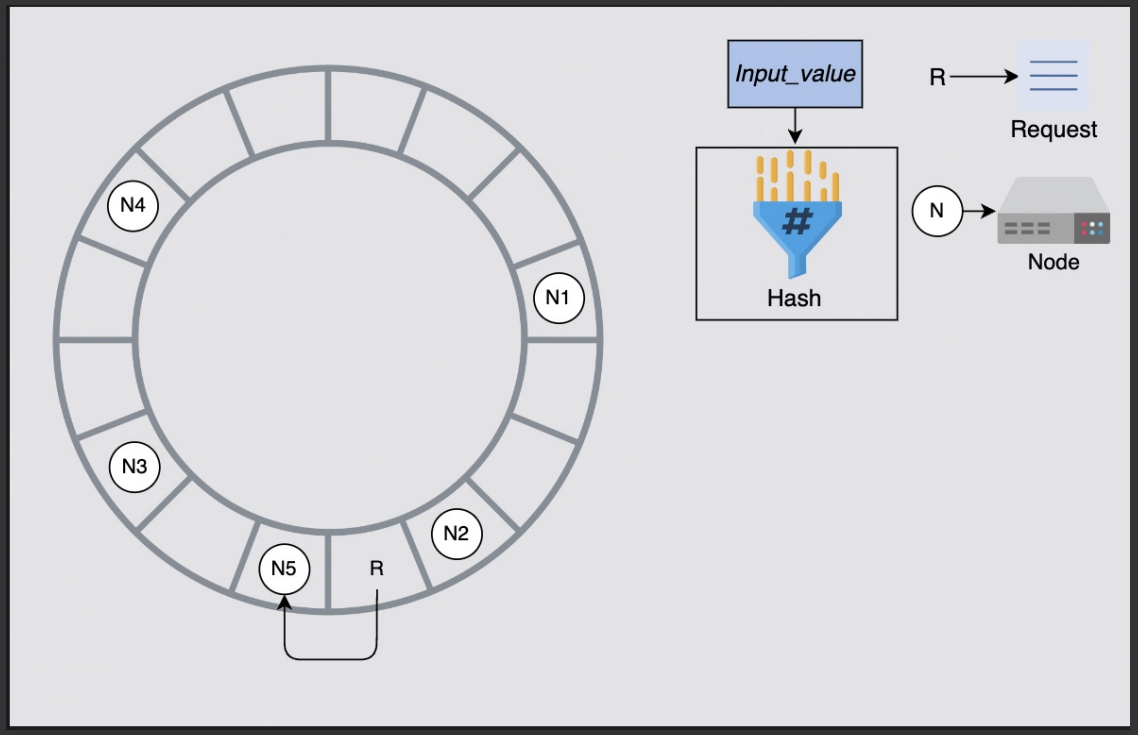


We want to add and remove nodes with minimal change in our infrastructure. But in this method, when we add or remove a node, we need to move a lot of keys. This is inefficient. For example, node 2 is removed, and suppose for the same key, the new server to process a request will be node 1 because
10%3=1. Nodes hold information in their local caches, like keys and their values. So, we need to move that request's data to the next node that has to process the request. But this replication can be costly and can cause high latency.

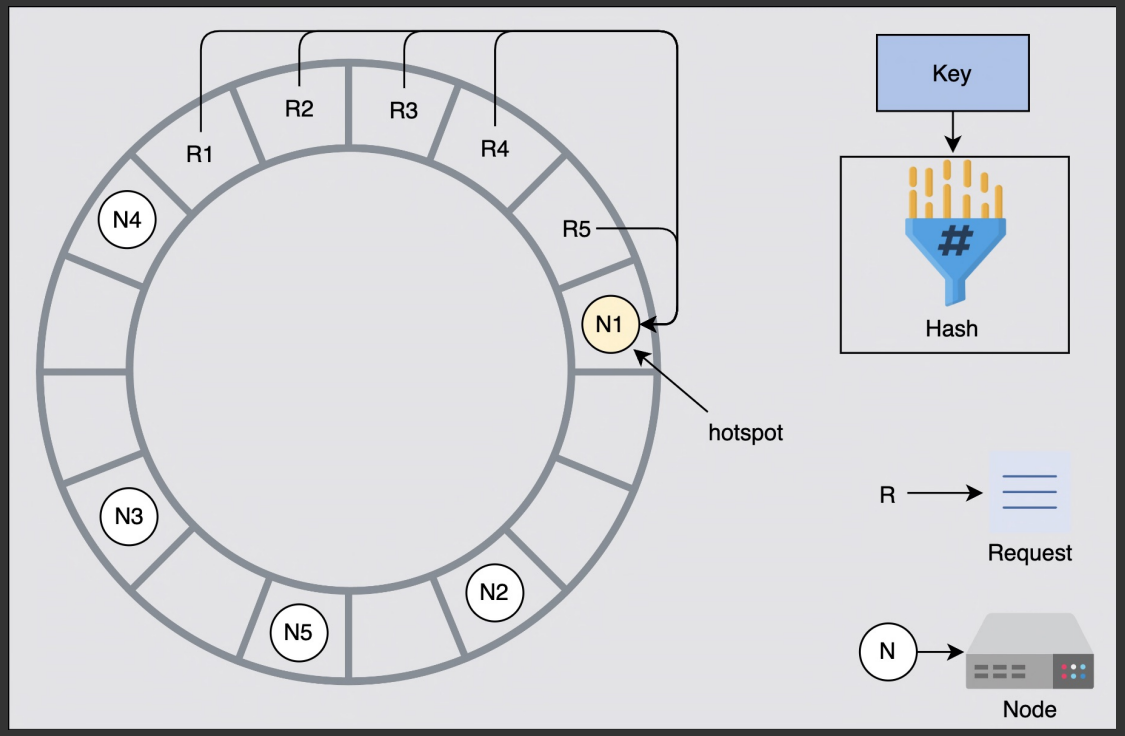
Why didn't we use load balancers to distribute the requests to all nodes?

Load balancers distribute client requests according to an algorithm. That algorithm can be as simple as explained above, or it can be something detailed, as described in the next section. The next method we'll discuss can be one of the ways the load balancers balance the requests across the nodes.

Consistent hashing
Consistent hashing is an effective way to manage the load over the set of nodes. In consistent hashing, we consider that we have a conceptual ring of hashes from 0 to n-1 where n is the number of available hash values. We use each node's ID, calculate its hash, and map it to the ring. We apply the same process to requests. Each request is completed by the next node that it finds by moving in the clockwise direction in the ring.

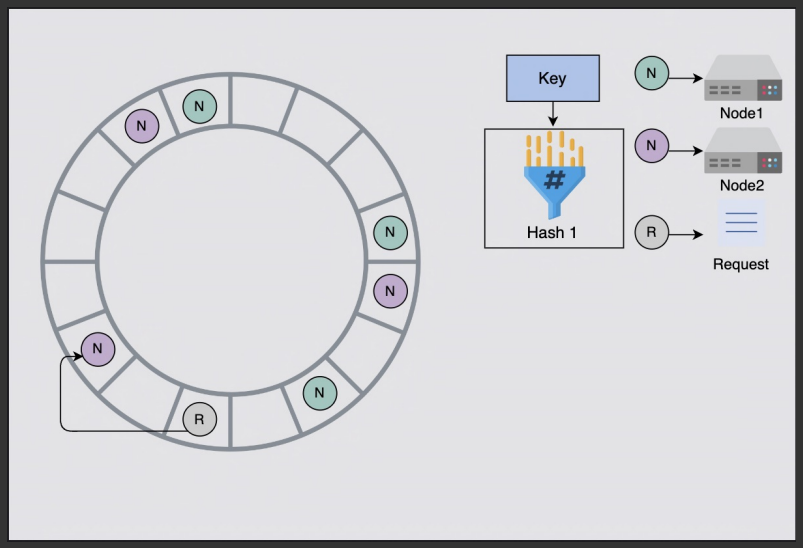


The primary benefit of consistent hashing is that as nodes join or leave, it ensures that a minimal number of keys need to move. However, the request load isn't equally divided in practice. Any server that handles a large chunk of data can become a bottleneck in a distributed system. That node will receive a disproportionately large share of data storage and retrieval requests, reducing the overall system performance. As a result, these are referred to as hotspots.



Hot spot condition in consistent hashing

Use virtual nodes
We'll use virtual nodes to ensure a more evenly distributed load across the nodes. Instead of applying a single hash function, we'll apply multiple hash functions onto the same key.



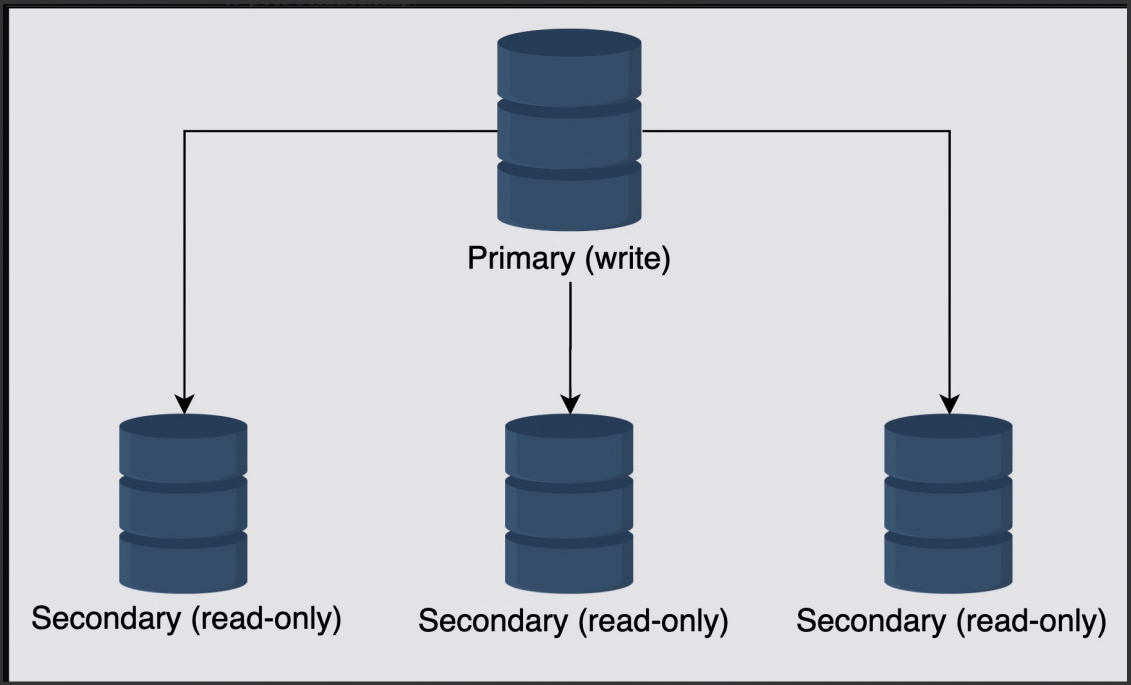
Advantages of virtual nodes
Following are some advantages of using virtual nodes:

If a node fails or does routine maintenance, the workload is uniformly distributed over other nodes. For each newly accessible node, the other nodes receive nearly equal load when it comes back online or is added to the system.

It's up to each node to decide how many virtual nodes it's responsible for, considering the heterogeneity of the physical infrastructure. For example, if a node has roughly double the computational capacity as compared to the others, it can take more load.

Data Replication

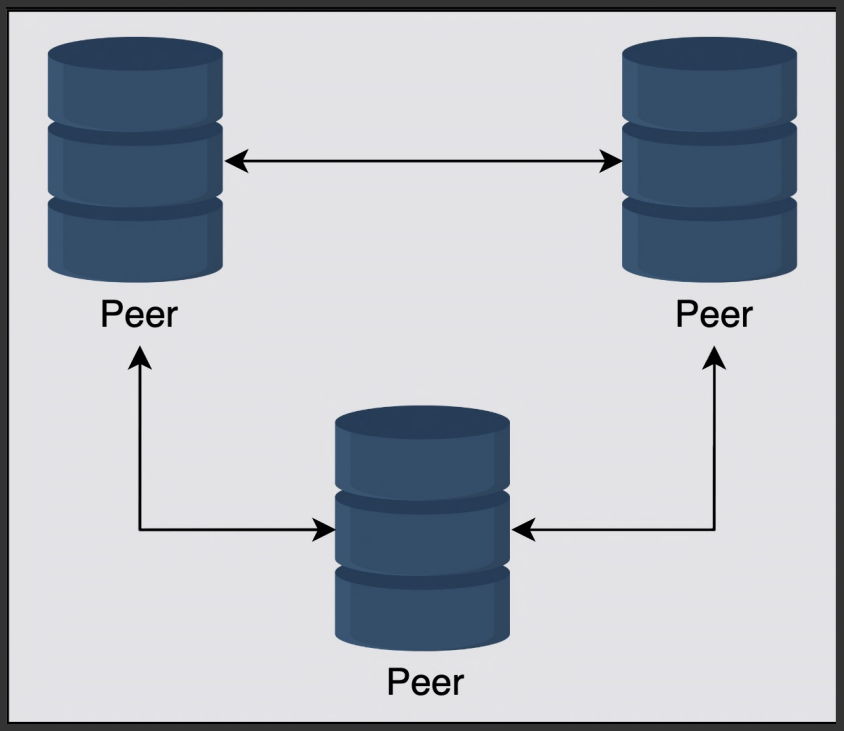
Primary-secondary approach
In a primary-secondary approach, one of the storage areas is primary, and other storage areas are secondary. The secondary replicates its data from the primary. The primary serves the write requests while the secondary serves read requests. After writing, there's a lag for replication. Moreover, if the primary goes down, we can't write into the storage, and it becomes a single point of failure.



Does the primary-secondary approach fulfill the requirements of the key-value store that we defined in the System Design: The Key-value Store lesson?

One of our requirements is that we need the ability to always write. This approach is good for the always read option. However, this approach doesn't include the ability to always write because it will overload the primary storage. Moreover, if a primary server fails, we need to upgrade a secondary to a primary. The availability of write will suffer as we won't allow writes during the switch-over time.

Peer-to-peer approach
In the peer-to-peer approach, all involved storage areas are primary, and they replicate the data to stay updated. Both read and write are allowed on all nodes. Usually, it's inefficient and costly to replicate in all n nodes. Instead, three or five is a common choice for the number of storage nodes to be replicated.



What is the impact of synchronous or asynchronous replication?

In synchronous replication, the speed of writing is slow because the data has to be replicated to all the nodes before acknowledging the user. It affects our availability, so we can't apply it. When we opt for asynchronous replication, it allows us to do speedy writes to the nodes.

In the context of the CAP theorem, key-value stores can either be consistent or be available when there are network partitions. For key-value stores, we prefer availability over consistency.

Versioning Data & Achieving Configurability

Data Versioning

When network partitions and node failures occur during an update, an object's version history might become fragmented. As a result, it requires a reconciliation effort on the part of the system. It's necessary to build a way that explicitly accepts the potential of several copies of the same data so that we can avoid the loss of any updates. It's critical to realize that some failure scenarios can lead to multiple copies of the same data in the system. So, these copies might be the same or divergent. Resolving the conflicts among these divergent histories is essential and critical for consistency purposes.

To handle inconsistency, we need to maintain causality between the events. We can do this using the timestamps and update all conflicting values with the value of the latest request. But time isn't reliable in a distributed system, so we can't use it as a deciding factor.

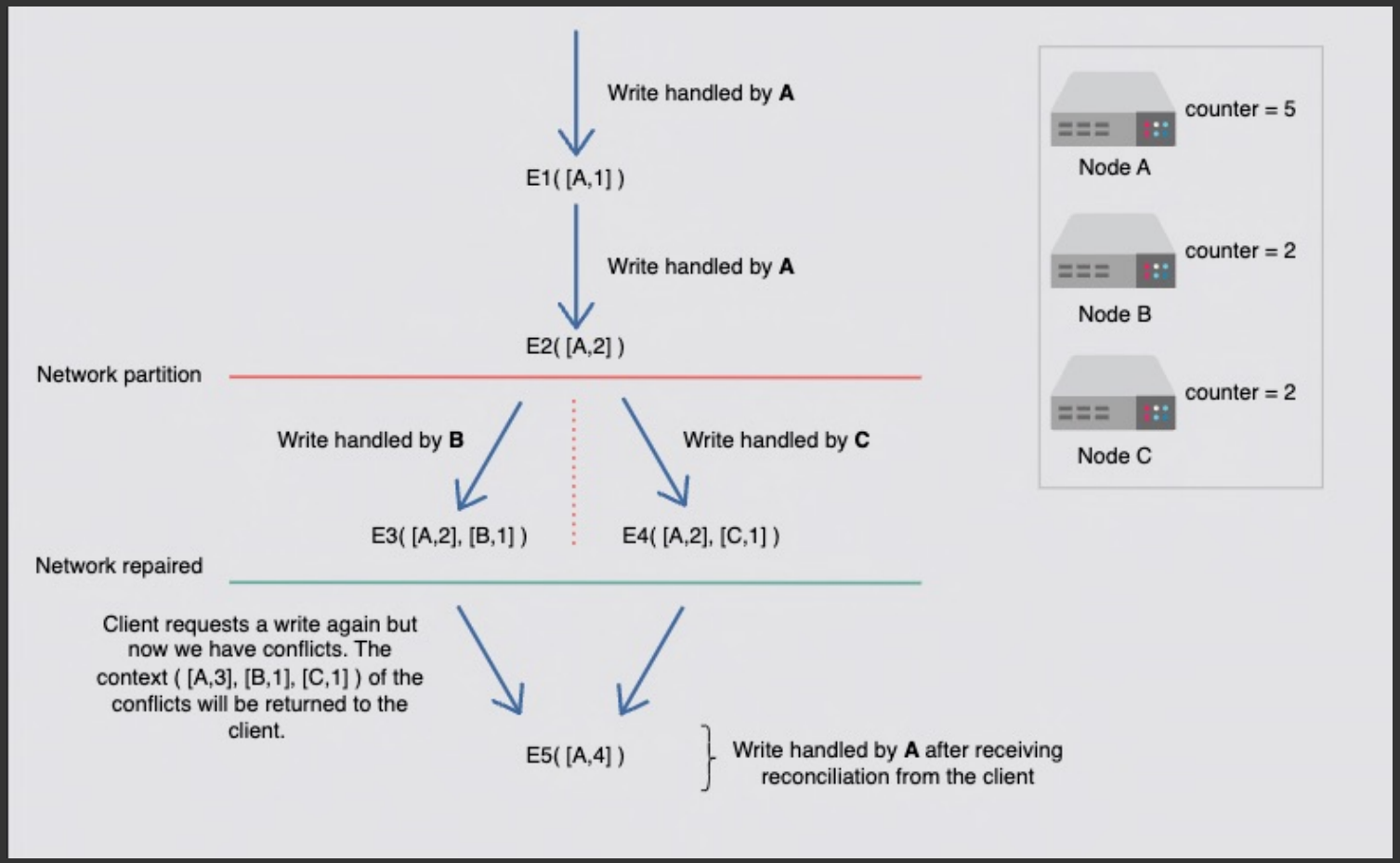
Another approach to maintaining causality effectively is by using vector clocks. A vector clock is a list of (node, counter) pairs. There's a single vector clock for every version of an object. If two objects have different vector clocks, we're able to tell whether they're causally related or not (more on this in a bit). Unless one of the two changes is reconciled, the two are deemed at odds.

→ We can modify the API design.

→ get(key)

→ put(key, context, value)

Vector clock Usage Example



Compromise with vector clocks limitations
The size of vector clocks may increase if multiple servers write to the same object simultaneously. It's unlikely to happen in practice because writes are typically handled by one of the top n nodes in a preference list.

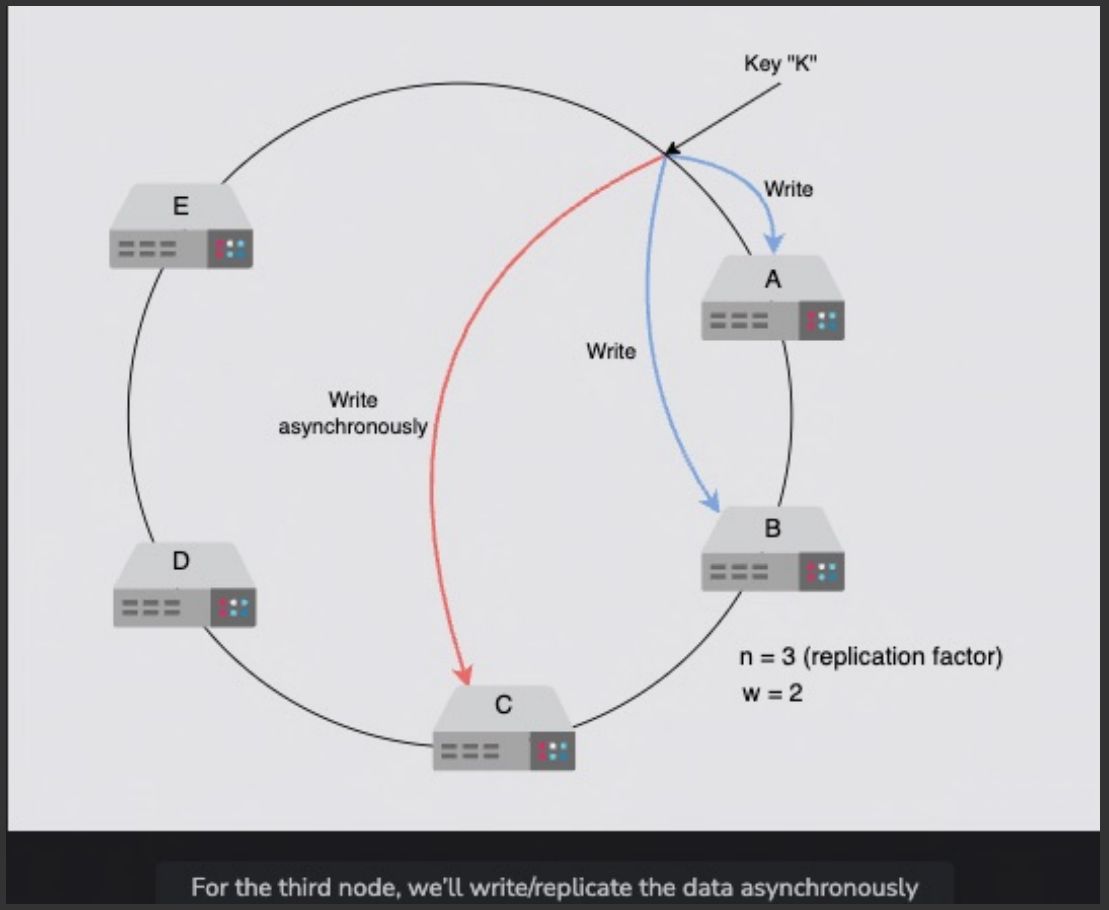
The get and put operations
One of our functional requirements is that the system should be configurable. We want to control the trade-offs between availability, consistency, cost-effectiveness, and performance. So, let's achieve configurability by implementing the basic get and put functions of the key-value store

Every node can handle the get (read) and put (write) operations in our system. A node handling a read or write operation is known as a coordinator. The coordinator is the first among the top n nodes in the preference list.

There can be two ways for a client to select a node:
We route the request to a generic load balancer.
We use a partition-aware client library that routes requests directly to the appropriate coordinator nodes.

Value Effects on Reads and Writes

n	r	w	Description
3	2	1	It won't be allowed as it violates our constraint $r + w > n$.
3	2	2	It will be allowed as it fulfills constraints.
3	3	1	It will provide speedy writes and slower reads since readers need to go to all n replicas for a value.
3	1	3	It will provide speedy reads from any node but slow writes since we now need to write to all n nodes synchronously.



For the third node, we'll write/replicate the data asynchronously

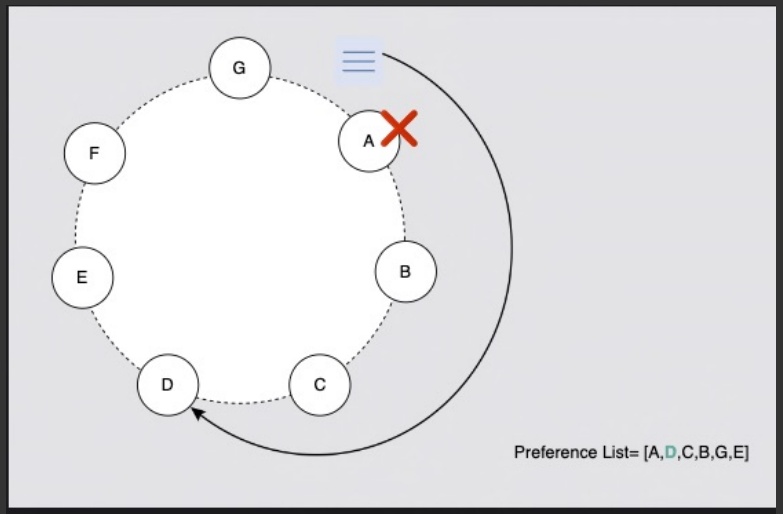
Enable Fault Tolerance & Failure Detection

Handling Temporary failure

→ A quorum is a minimum number of votes required for a distributed transaction to proceed with an operation.

→ We'll use a sloppy quorum instead of strict quorum membership. Usually, a leader manages the communication among the participants of the consensus. The participants send an acknowledgment after committing a successful write

→ In the sloppy quorum, the first n healthy nodes from the preference list handle all read and write operations. The n healthy nodes may not always be the first n nodes discovered when moving clockwise in the consistent hash ring.



Request is sent to node D as node A is down

Let's consider the following configuration with $n = 3$. If node A is briefly unavailable or unreachable during a write operation, the request is sent to the next healthy node from the preference list, which is node D in this case. It ensures the desired availability and durability. After processing the request, the node D includes a hint as to which node was the intended receiver (in this case, A). Once node A is up and running again, node D sends the request information to A so it can update its data. Upon completion of the transfer, D removes this item from its local storage without affecting the total number of replicas in the system.

This approach is called a **hinted handoff**. Using it, we can ensure that reads and writes are fulfilled if a node faces temporary failure.

What are the limitations of using hinted handoff?

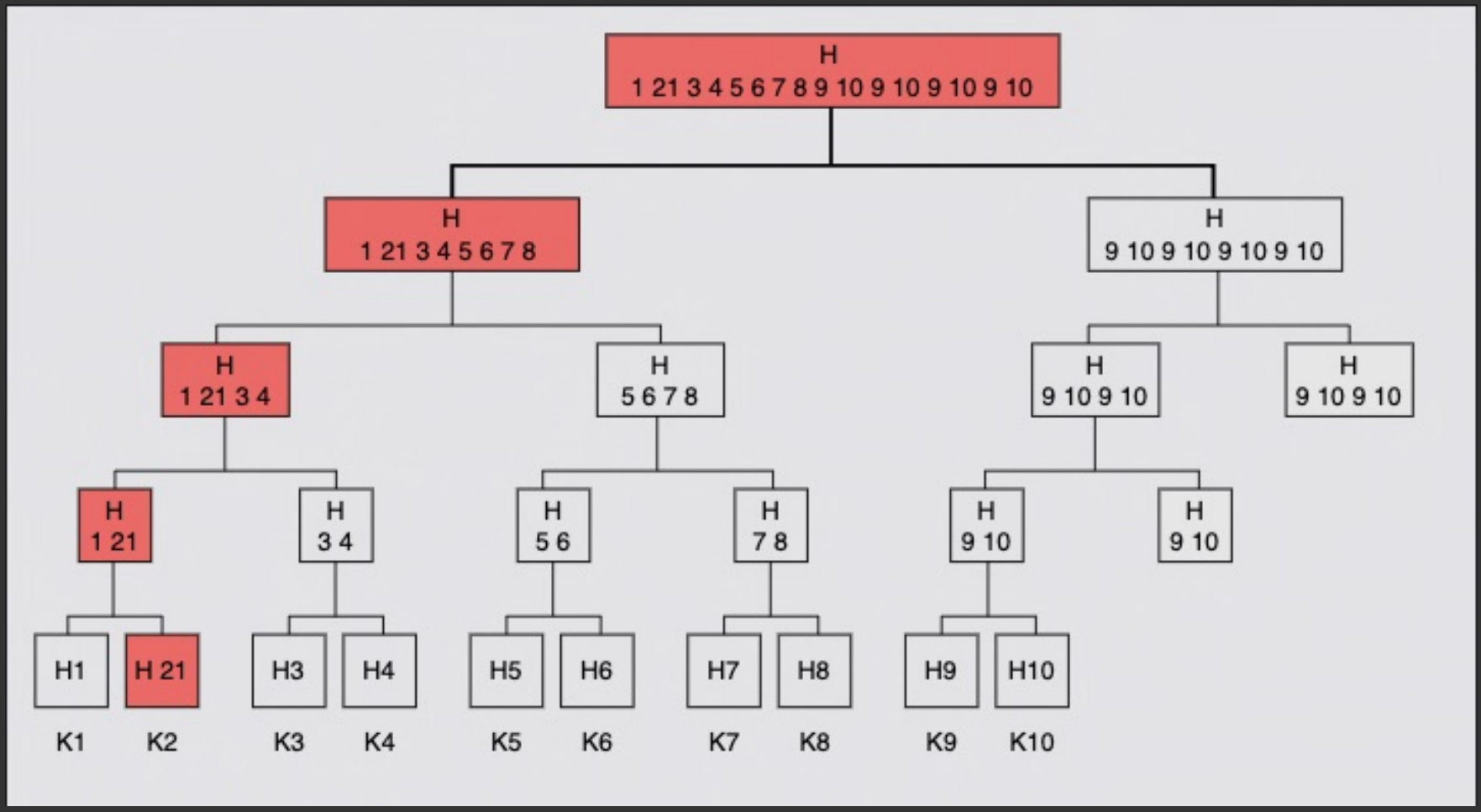
Hide Answer

A minimal churn in system membership and transient node failures are ideal for hinted handoff. However, hinted replicas may become unavailable before being restored to the originating replica node in certain circumstances.

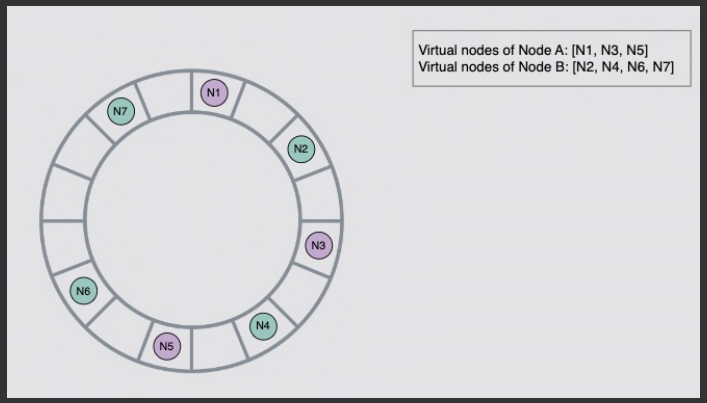
Handling Permanent Failure

In a **Merkle tree**, the values of individual keys are hashed and used as the leaves of the tree. There are hashes of their children in the parent nodes higher up the tree. Each branch of the Merkle tree can be verified independently without the need to download the complete tree or the entire dataset. While checking for inconsistencies across copies, Merkle trees reduce the amount of data that must be exchanged. There's no need for synchronization if, for example, the hash values of two trees' roots are the same and their leaf nodes are also the same. Until the process reaches the tree leaves, the hosts can identify the keys that are out of sync when the nodes exchange the hash values of children. The Merkle tree is a mechanism to implement anti-entropy, which means to keep all the data consistent. It reduces data transmission for synchronization and the number of discs accessed during the anti-entropy process.

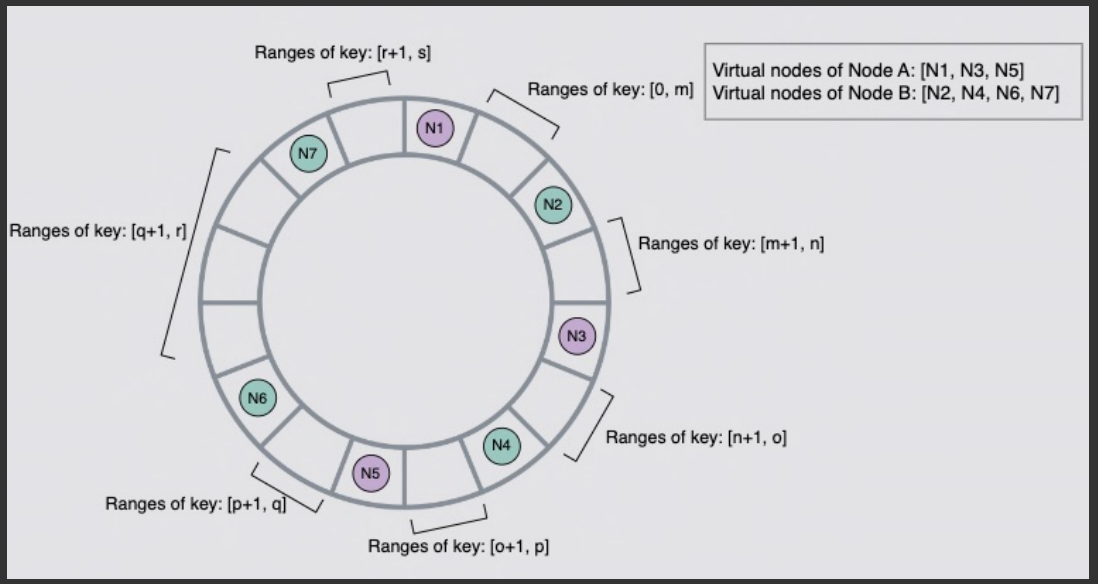
The following slides explain how Merkle trees work:


H
1 21 3 4 5 6 7 8 9 10 9 10 9 10 9 10
H
1 21 3 4 5 6 7 8
H
9 10 9 10 9 10 9 10
H
1 21 3 4
H
5 6 7 8
H
9 10 9 10
H
9 10 9 10
H
1 21
H
3 4
H
5 6
H
7 8
H
9 10
H
9 10
H1
H 21
H3
H4
H5
H6
H7
H8
H9
H10
K1
K2
K3
K4
K5
K6
K7
K8
K9
K10

How does Merkle trees work



Step 1

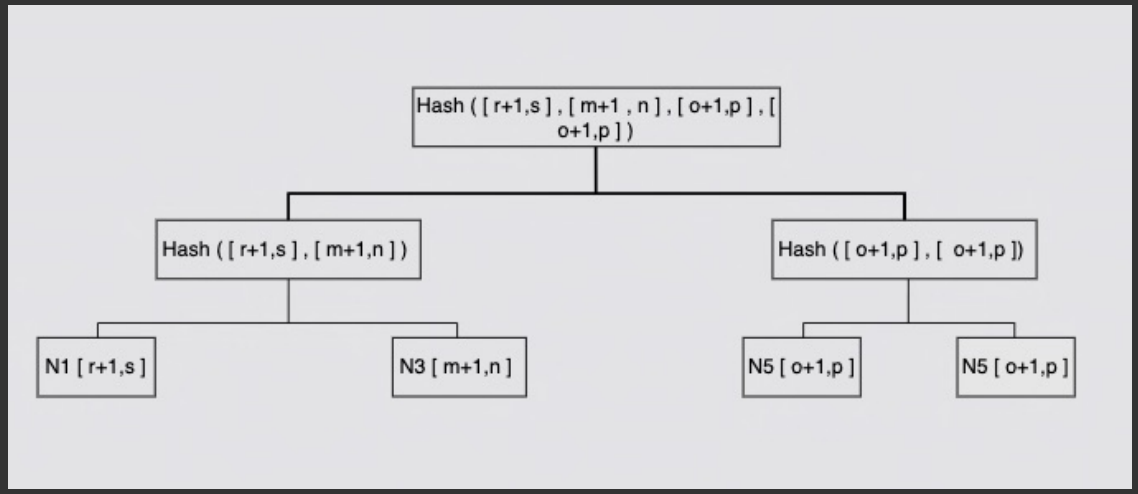


Step 2

Node A	
Virtual node	Range
N1	[r+1, s]
N3	[m+1, n]
N5	[o+1, p]

Node B	
Virtual node	Range
N2	[0,m]
N4	[n+1, o]
N6	[p+1, q]
N7	[q+1, r]

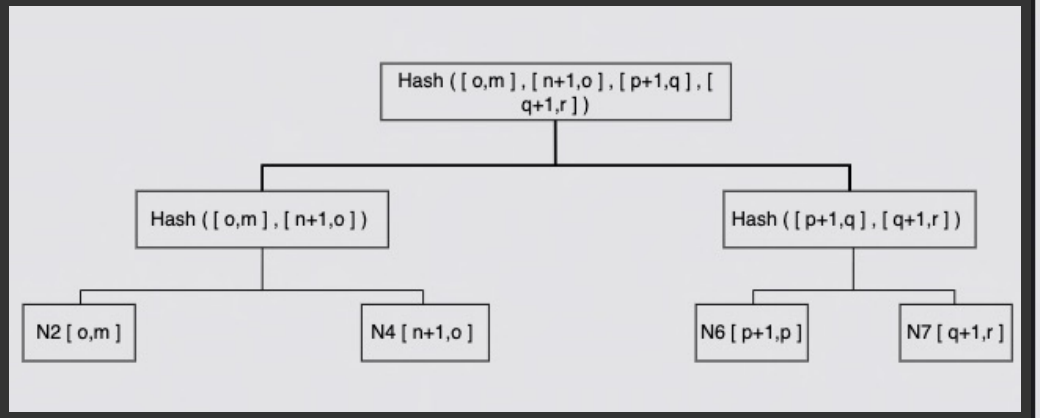
Step 3



Step 4

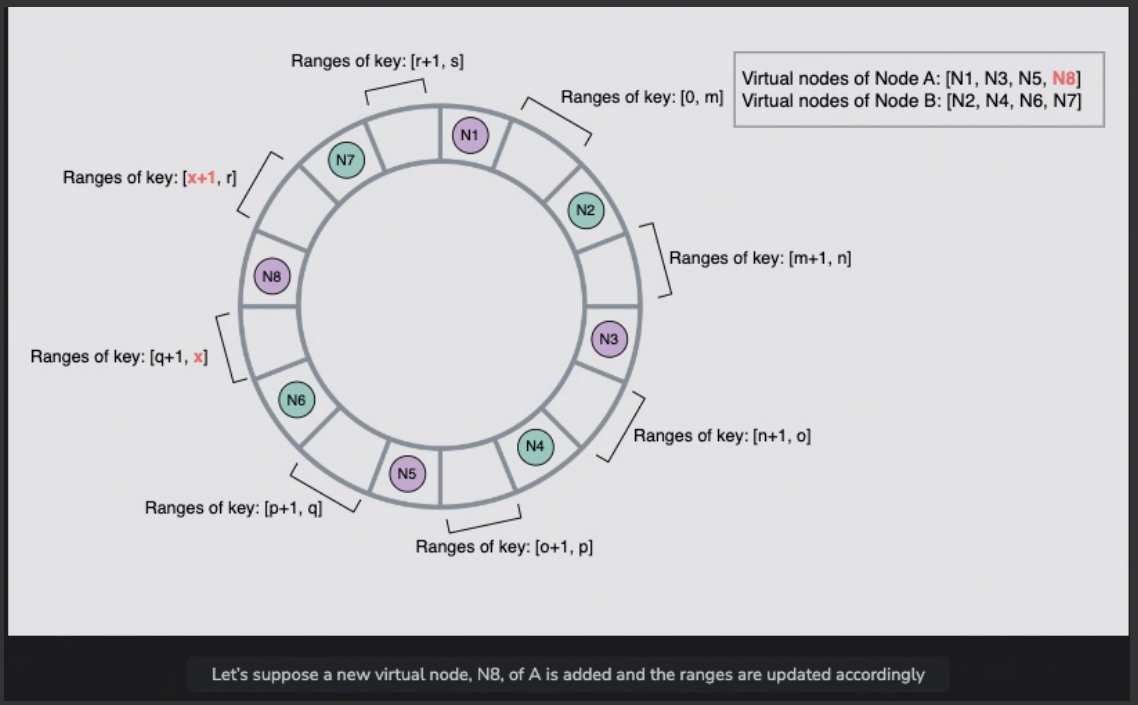
↓

for Node A



Step 5

↓ for Node B

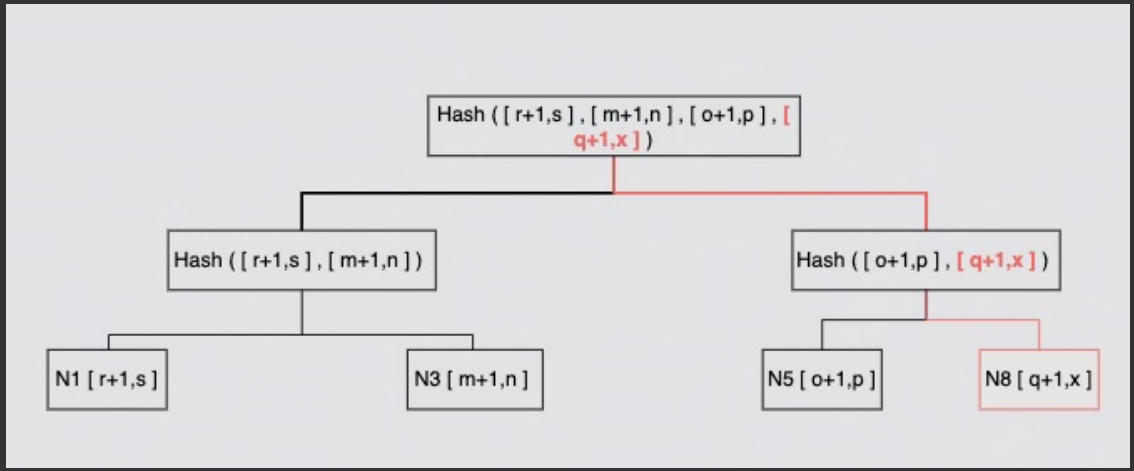


Step 6

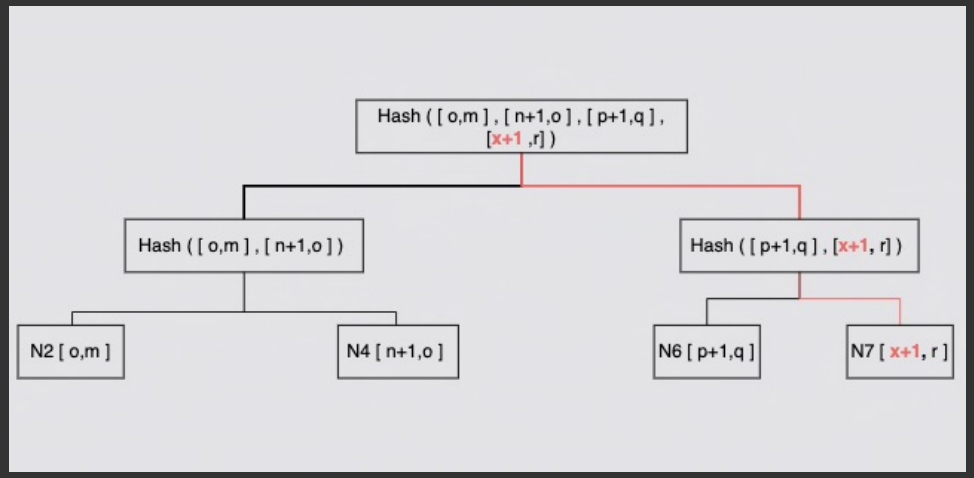
Node A	
Virtual node	Range
N1	[r+1, s]
N3	[m+1, n]
N5	[o+1, p]
N8	[q+1, x]

Node B	
Virtual node	Range
N2	[0,m]
N4	[n+1, o]
N6	[p+1, q]
N7	[x+1, r]

Step 7.



Step 8 . ↓ for Node A



→ for Node B

Step 9.

The advantage of using Merkle trees is that each branch of the Merkle tree can be examined independently without requiring nodes to download the tree or the complete dataset. It reduces the quantity of data that must be exchanged for synchronization and the number of disc accesses that are required during the anti-entropy procedure.

The disadvantage is that when a node joins or departs the system, the tree's hashes are recalculated because multiple key ranges are affected.

Promote Membership in Ring to detect Failure

A gossip-based protocol also maintains an eventually consistent view of membership. When two nodes randomly choose one another as their peer, both nodes can efficiently synchronize their persisted membership histories.

Let's learn how a gossip-based protocol works by considering the following example. Say node A starts up for the first time, and it randomly adds nodes B and E to its token set. The token set has virtual nodes in the consistent hash space and maps nodes to their respective token sets. This information is stored locally on the disk space of the node.

Now, node A handles a request that results in a change, so it communicates this to B and E. Another node, D, has C and E in its token set. It makes a change and tells C and E. The other nodes do the same process. This way, every node eventually knows about every other node's information. It's an efficient way to share information asynchronously, and it doesn't take up a lot of bandwidth.

Keeping in mind our consistent hashing approach, can the gossip-based protocol fail?

Hide Answer

Yes, the gossip-based protocol can fail. For example, the virtual node, $N1$, of node A wants to be added to the ring. The administrator asks $N2$, which is also a virtual node of A. In such a case, both nodes consider themselves to be part of the ring and won't be aware that they're the same server. If any change is made, it will keep on updating itself, which is wrong. This is called **logical partitioning**.

The gossip-based protocol works when all the nodes in the ring are connected in a single graph (i.e., have one connected component in the graph). That implies that there is a path from any node to any other node (possibly via different intermediaries). Different issues such as high churn (coming and going of nodes), issues with virtual node to physical node mappings, etc. can create a situation that is the same as if the real network had partitioned some nodes from the rest and now updates from one set won't reach to the other. Therefore just having a gossip protocol in itself is not sufficient for proper information dissemination; keeping the topology in a good, connected state is also necessary.

How can we prevent logical partitioning?

Hide Answer

We can make a few nodes play the role of seeds to avoid logical partitions. We can define a set of nodes as seeds via a configuration service. This set of nodes is known to all the working nodes since they can eventually reconcile their membership with a seed. So, logical partitions are pretty rare.