Enable Fault Tolerance and Failure Detection

Learn how to make a key-value store fault tolerant and able to detect failure.

We'll cover the following

Handle temporary failures

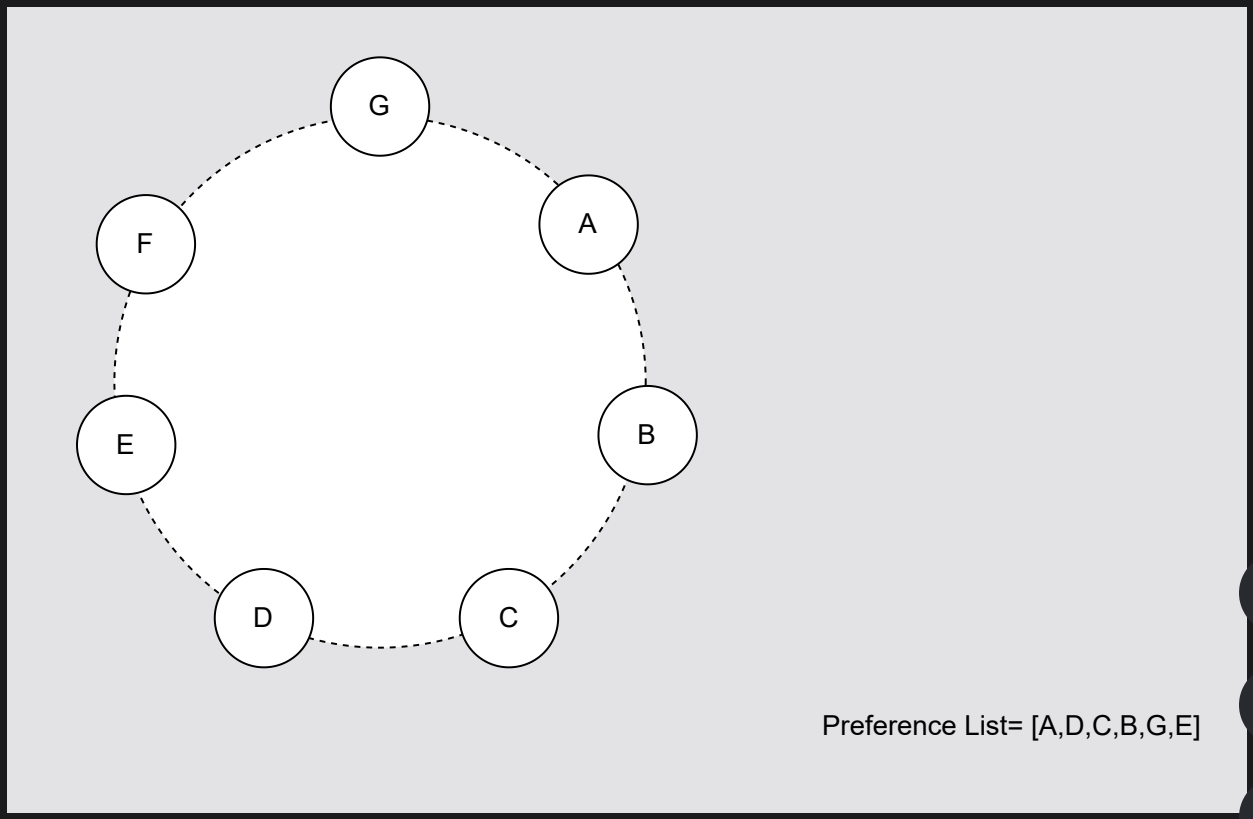
Typically, distributed systems use a quorum-based approach to handle failures. A quorum is the minimum number of votes required for a distributed transaction to proceed with an operation. If a server is part of the consensus and is down, then we can’t perform the required operation. It affects the availability and durability of our system.

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. Upon receiving these acknowledgments, the leader responds to the client. However, the drawback is that the participants are easily affected by the network outage. If the leader is temporarily down and the participants can’t reach it, they declare the leader dead. Now, a new leader has to be reelected. Such frequent elections have

a negative impact on performance because the system spends more time picking a leader than accomplishing any actual work.

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.

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.



Suppose we have seven nodes in our ring and a preference list of the nodes

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

> **Note**: A highly available storage system must handle data center failure due to power outages, cooling failures, network failures, or natural disasters. For this, we should ensure replication across the data centers. So, if one data center is down, we can recover it from the other.

Point to Ponder

Question

What are the limitations of using hinted handoff?

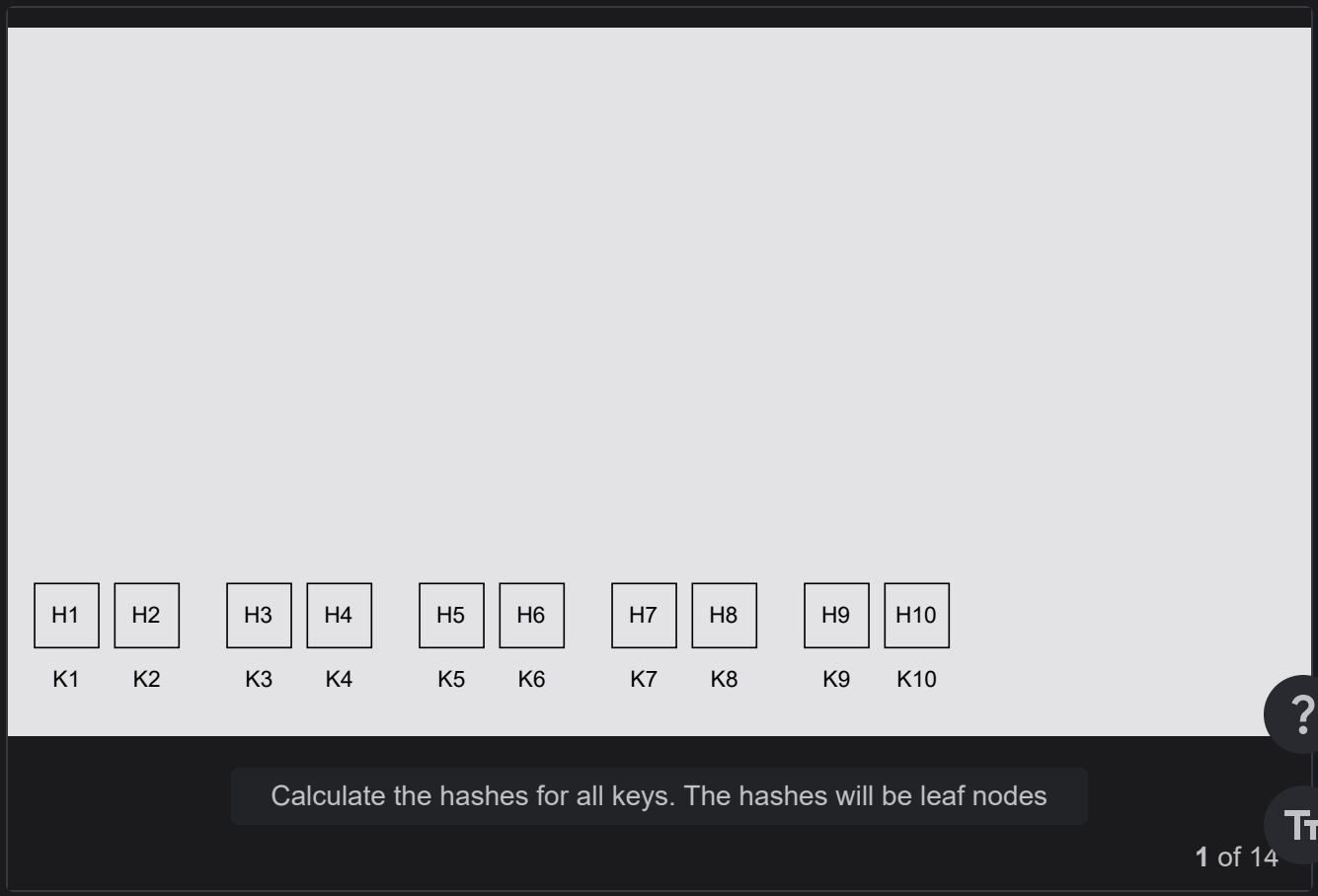
Show Answer

Handle permanent failures

In the event of permanent failures of nodes, we should keep our replicas synchronized to make our system more durable. We need to speed up the detection of inconsistencies between replicas and reduce the quantity of transferred data. We'll use Merkle trees for that.

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:



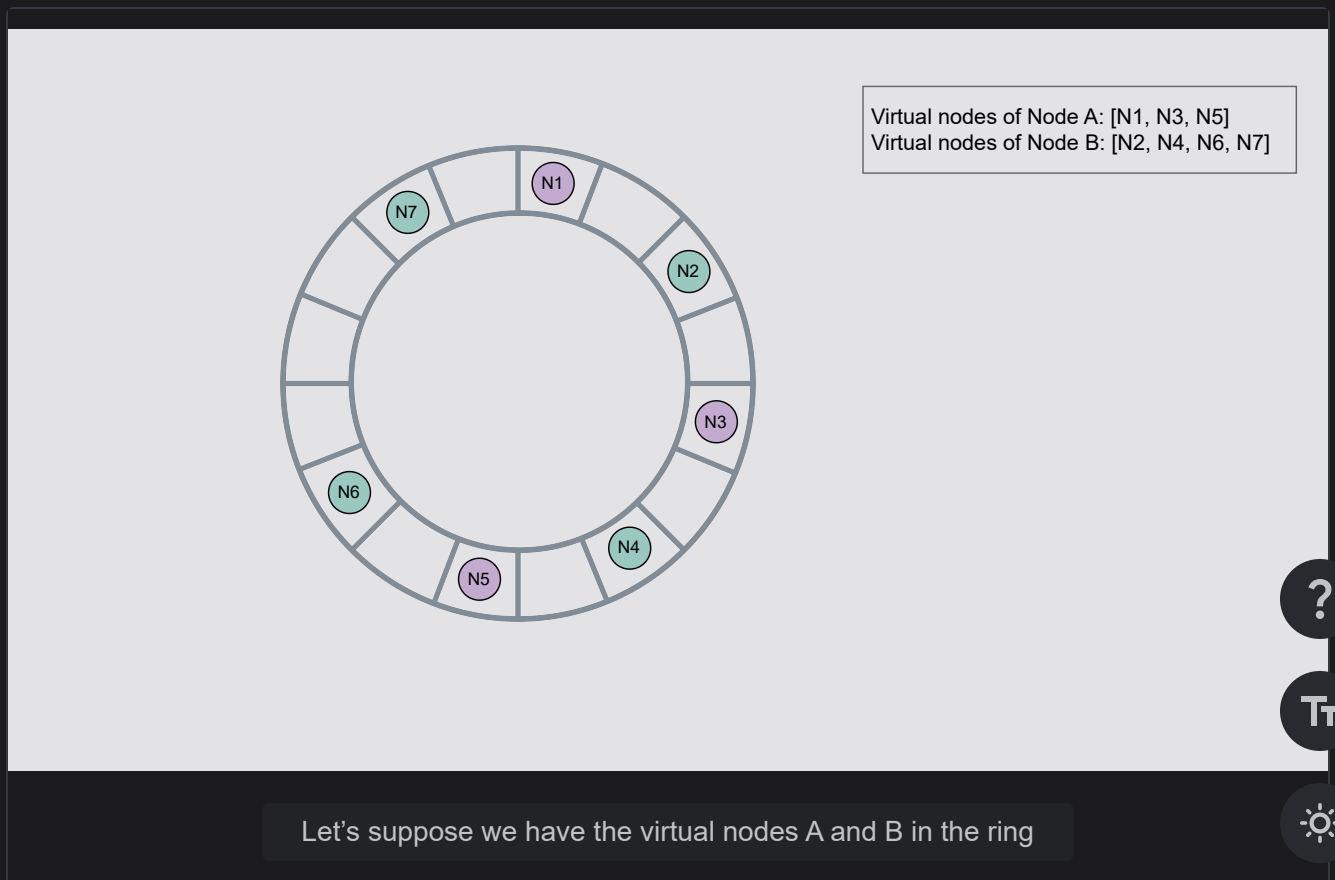
Anti-entropy with Merkle trees

Each node keeps a distinct Merkle tree for the range of keys that it hosts for each virtual node. The nodes can determine if the keys in a given range are correct. The root of the Merkle tree corresponding to the common key ranges is exchanged between two nodes. We'll make the following comparison:

1. Compare the hashes of the root node of Merkle trees.
2. Do not proceed if they're the same.
3. Traverse left and right children using recursion. The nodes identify whether or not they have any differences and perform the necessary synchronization.

The following slides explain more about how Merkle trees work.

> **Note**: We assume the ranges defined are hypothetical for illustration purposes.



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.

We want our nodes to detect the failure of other nodes in the ring, so let's see how we can add it to our proposed design.

Promote membership in the ring to detect failures

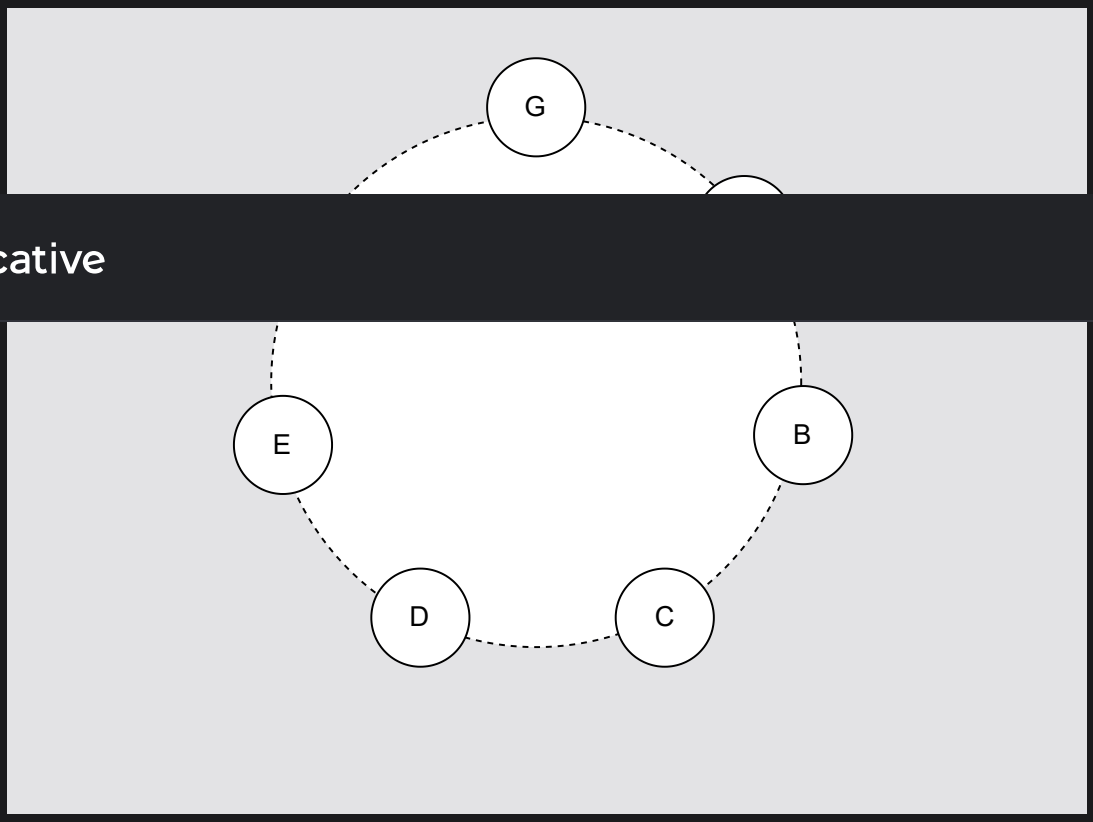
The nodes can be offline for short periods, but they may also indefinitely go offline. We shouldn't rebalance partition assignments or fix unreachable replicas when a single node goes down because it's rarely a permanent departure. Therefore, the addition and removal of nodes from the ring should be done carefully.

Planned commissioning and decommissioning of nodes results in membership changes. These changes form history. They're recorded persistently on the storage for each node and reconciled among the ring members using a gossip protocol. 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.



A set of nodes in a ring

Points to Ponder

Question 1

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

Show Answer

1 of 2

Decentralized failure detection protocols use a gossip-based protocol that allows each node to learn about the addition or removal of other nodes. The join and leave methods of the arriving or leaving nodes explicitly notify the other nodes about the permanent node additions and removals. The individual nodes detect temporary node failures when they fail to communicate with another node. If a node fails to communicate to any of the nodes present in its token set for the authorized time, then it communicates to the administrators that the node is dead.

Conclusion

A key-value store provides flexibility and allows us to scale the applications that have unstructured data. Web applications can use key-value stores to store information about a user's session and preferences. When using a user key, all the data is accessible, and key-value stores are ideal for rapid reads and write operations. Key-value stores can be used to power real-time recommendations and advertising because the stores can swiftly access and present fresh recommendations.