CORE SYSTEM DESIGN PRINCIPLES

Availability

What do we mean by Availability?

Availability refers to the ability of a system to provide its services to clients even in the presence of failures. This is often measured in terms of the percentage of time that the system is up and running, also known as its uptime.

It is usually expressed as a percentage, indicating the system's uptime over a specific period.

The formal definition of availability is:

Availability = Uptime / (Uptime + Downtime)

Uptime: The period during which a system is functional and accessible.

Downtime: The period during which a system is unavailable due to failures, maintenance, or other issues.

The Nine's of availability

Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. It is generally measured in the number of 9s.

Availability = Uptime(Uptime+Downtime)

If availability is 99.00% available, it is said to have "2 nines" of availability, and if it is 99.9%, it is called "3 nines", and so on.

Availability (Percent)	Downtime (Year)	Downtime (Month)	Downtime (Week)
90% (one nine)	36.53 days	72 hours	16.8 hours
99% (two nines)	3.65 days	7.20 hours	1.68 hours
99.9% (three nines)	8.77 hours	43.8 minutes	10.1 minutes
99.99% (four nines)	52.6 minutes	4.32 minutes	1.01 minutes
99.999% (five nines)	5.25 minutes	25.9 seconds	6.05 seconds
99.9999% (six nines)	31.56 seconds	2.59 seconds	604.8 milliseconds
99.99999% (seven nines)	3.15 seconds	263 milliseconds	60.5 milliseconds
99.999999% (eight nines)	315.6 milliseconds	26.3 milliseconds	6 milliseconds
99.9999999% (nine nines)	31.6 milliseconds	2.6 milliseconds	0.6 milliseconds

Availability in Sequence vs Parallel

If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.

Sequence

- Overall availability decreases when two components are in sequence.
- Availability (Total)=Availability (Foo)∗Availability (Bar)
- For example, if both Foo and Bar each had 99.9% availability, their total availability in sequence would be 99.8%.

Parallel

- Overall availability increases when two components are in parallel.
- Availability (Total)=1−(1−Availability (Foo))∗(1−Availability (Bar))
- For example, if both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

Availability vs. Reliability

If a system is reliable, it is available. However, if it is available, it is not necessarily reliable. In other words, high reliability contributes to high availability, but it is possible to achieve high availability even with an unreliable system.

High availability vs. Fault Tolerance

Both high availability and fault tolerance apply to methods for providing high uptime levels. However, they accomplish the objective differently.

A fault-tolerant system has no service interruption but a significantly higher cost, while a highly available system has minimal service interruption. Fault-tolerance requires full hardware redundancy as if the main system fails, with no loss in uptime, another system should take over.

Availability Patterns

There are two complementary patterns to support high availability: fail-over and replication.

1. FailOver

Failover is an availability pattern that is used to ensure that a system can continue to function in the event of a failure. It involves having a backup component or system that can take over in the event of a failure.

In a failover system, there is a primary component that is responsible for handling requests, and a secondary (or backup) component that is on standby. The primary component is monitored for failures, and if it fails, the secondary component

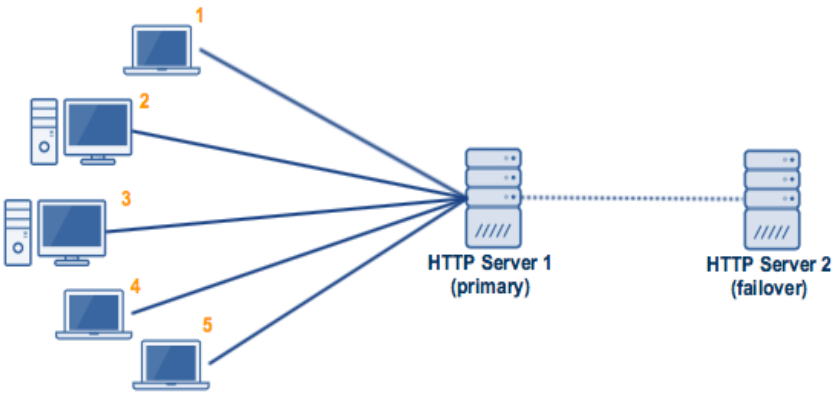
is activated to take over its duties. This allows the system to continue functioning with minimal disruption.

Failover can be implemented in various ways, such as active-passive, active-active, and hot-standby.

2. Active-Passive Redundancy

An active-passive architectural pattern consists of at least two nodes. The passive server (failover) acts as a backup that remains on standby and takes over in the event the active server gets disconnected for whatever reason. The primary active server hosts production, test and development applications.

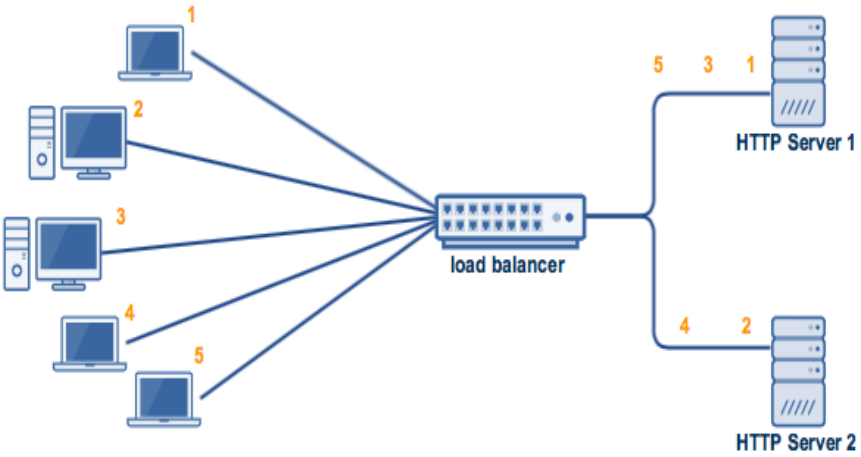
The secondary passive server essentially remains dormant during normal operation. A major disadvantage of this model is that there is no guarantee that the production application will function as expected on the passive server. The model is also considered a relatively wasteful approach because expensive hardware is left unused.



Active-Active Redundancy

The active-active model also contains at least two nodes; however, in this architectural pattern, multiple nodes are actively running the same services simultaneously. In order to fully utilize all the active nodes, an active-active cluster uses load balancing to distribute workloads across the nodes in order to prevent any single node from being overloaded. The distributed workload subsequently leads to a marked improvement in response times and throughput.

The load balancers uses a set of complex algorithms to assign clients to the nodes, the connections are typically based on performance metrics and health checks. In order to guarantee seamless operability, all the nodes in the cluster must be configured for redundancy. A potential drawback for an active-active redundancy is that in case one of the nodes fails, client sessions might be dropped, forcing them to re-login into the system. However, this can easily be mitigated by ensuring that the individual configuration settings of each node are virtually identical.



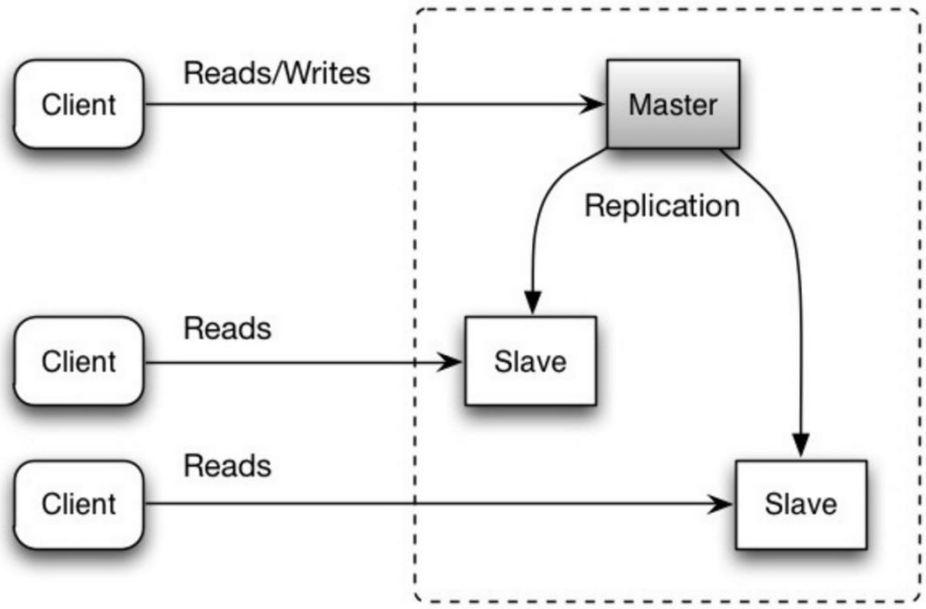
Replication

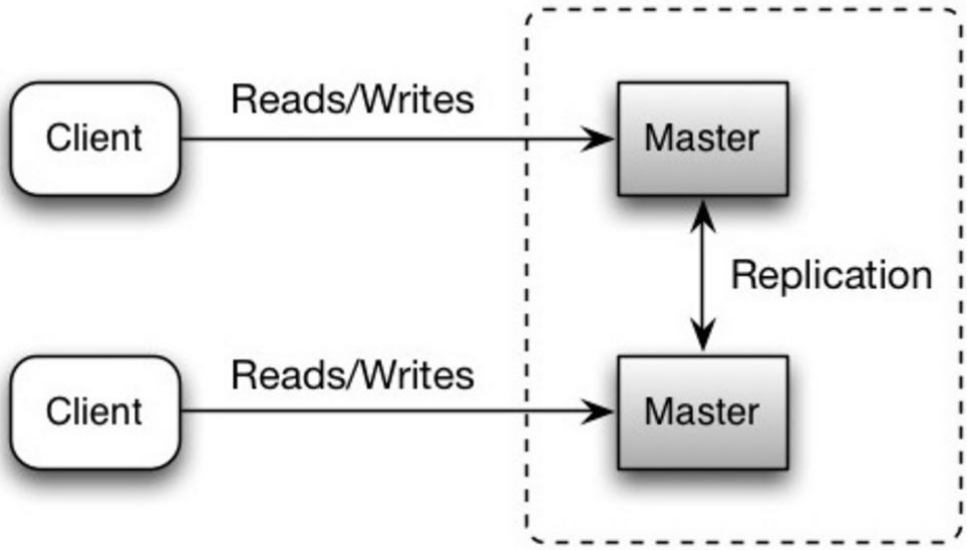
Replication is an availability pattern that involves having multiple copies of the same data stored in different locations. In the event of a failure, the data can be retrieved from a different location. There are two main types of replication: Master-Master replication and Master-Slave replication.

Master-slave replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.

Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.





Disadvantages: master-master replication

- You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
- Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
- Conflict resolution comes more into play as more write nodes are added and as latency increases.
- Additional logic is needed to promote a slave to a master.

Disadvantages: replication

- There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
- Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
- The more read slaves, the more you have to replicate, which leads to greater replication lag.

- On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
- Replication adds more hardware and additional complexity.

Consistency

What do we mean by consistency?

Consistency refers to the system's ability to ensure that all users see the same data, regardless of where or when they access it. In a consistent system, any update to the data is immediately visible to all users, and there are no conflicting or outdated versions of the data.

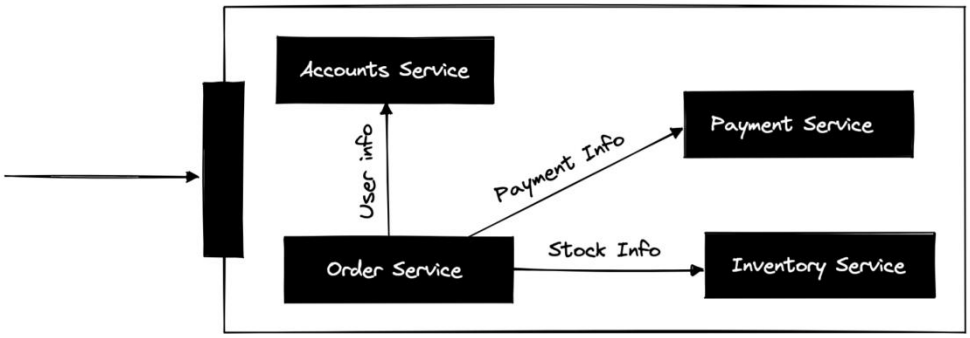
Consistency Patterns in Distributed Systems

Before we talk about the Consistency Patterns, we should know what a distributed system is. Simply put, a distributed system is a system that consists of more than one components, and each component is responsible for one part of the application.

A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal.

Example of a Distributed System

Imagine we have an e-commerce application where we are selling books. This application may consist of multiple different components. For example, one server might be responsible for the accounts, another might be responsible for the payments, one might be responsible for storing orders, one might be responsible for loyalty points and relevant functionalities, and another might be responsible for maintaining the books inventory and so on.



Now, if a user buys a book, there might be different services involved in placing the order; order service for storing the order, payment service for handling the payments, and inventory service for keeping the stock of that ordered book up to date. This is an example of a distributed system, an application that consists of multiple different components, each of which is responsible for a different part of the application.

Why is Consistency Important?

When working with distributed systems, we need to think about managing the data across different servers. If we take the above example of the e-commerce application, we can see that the inventory service must have up-to-date stock information for the ordered items if the user places an order. Now, there might be two different users looking at the same book. Now imagine if one of the customers places a successful order, and before the inventory service can update the stock, the second customer also places the order for the same book. In that case, when the inventory wasn't updated, we will have the wrong stock information when the second order was placed, i.e., the ordered book may or may not be available in stock. This is where different consistency patterns come into play. They help ensure that the data is consistent across the application.

Consistency Patterns

Consistency patterns refer to the ways in which data is stored and managed in a distributed system and how that data is made available to users and applications.

There are three main types of consistency patterns:

1. **Strong consistency**
2. **Weak consistency**
3. **Eventual Consistency**

Each of these patterns has its own advantages and disadvantages, and the choice of which pattern to use will depend on the specific requirements of the application or system.

1. Strong Consistency

After an update is made to the data, it will be immediately visible to any subsequent read operations. The data is replicated in a synchronous manner, ensuring that all copies of the data are updated at the same time.

In a strong consistency system, any updates to some data are immediately propagated to all locations. This ensures that all locations have the same version of the data, but it also means that the system is not highly available and has high latency.

An example of strong consistency is a financial system where users can transfer money between accounts. The system is designed for high data integrity, so the data is stored in a single location and updates to that data are immediately propagated to all other locations. This ensures that all users and applications are working with the same, accurate data. For instance, when a user initiates a transfer of funds from one account to another, the system immediately updates the balance of both accounts and all other system components are immediately aware of the change. This ensures that all users can see the updated balance of both accounts and prevents any discrepancies.

2. Weak Consistency

After an update is made to the data, it is not guaranteed that any subsequent read operation will immediately reflect the changes made. The read may or may not see the recent write.

In a weakly consistent system, updates to the data may not be immediately propagated. This can lead to inconsistencies and conflicts between different versions of the data, but it also allows for high availability and low latency.

Another example of weak consistency is a gaming platform where users can play online multiplayer games. When a user plays a game, their actions are immediately visible to other players in the same data center, but if there was a lag or temporary connection loss, the actions may not be seen by some of the users and the game will continue. This can lead to inconsistencies between different versions of the game state, but it also allows for a high level of availability and low latency.

3. Eventual Consistency

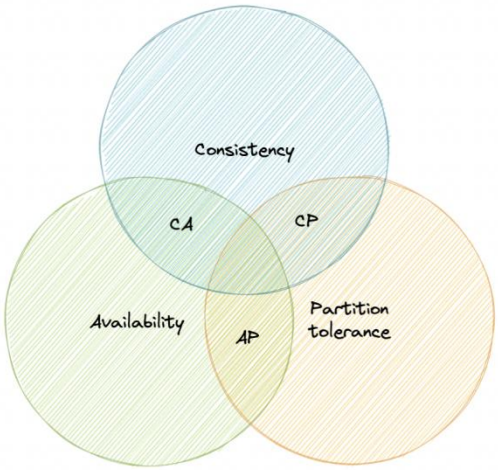
Eventual consistency is a form of Weak Consistency. After an update is made to the data, it will be eventually visible to any subsequent read operations. The data is replicated in an asynchronous manner, ensuring that all copies of the data are eventually updated.

In an eventually consistent system, data is typically stored in multiple locations, and updates to that data are eventually propagated to all locations. This means that the system is highly available and has low latency, but it also means that there may be inconsistencies and conflicts between different versions of the data.

An example of eventual consistency is a social media platform where users can post updates, comments, and messages. The platform is designed for high availability and low latency, so the data is stored in multiple data centers around the world. When a user posts an update, the update is immediately visible to other users in the same data center, but it may take some time for the update to propagate to other data centers. This means that some users may see the update while others may not, depending on which data center they are connected to. This can lead to inconsistencies between different versions of the data, but it also allows for a high level of availability and low latency.

CAP Theorem

The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data.), you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed. However, as you will see below, you don't have as many options here as you might think.



Consistency

Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated across all the nodes in the system before the write is deemed "successful".

Availability

Availability means that any client making a request for data gets a response, even if one or more nodes are down.

Partition tolerance

Partition tolerance means the system continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of

network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

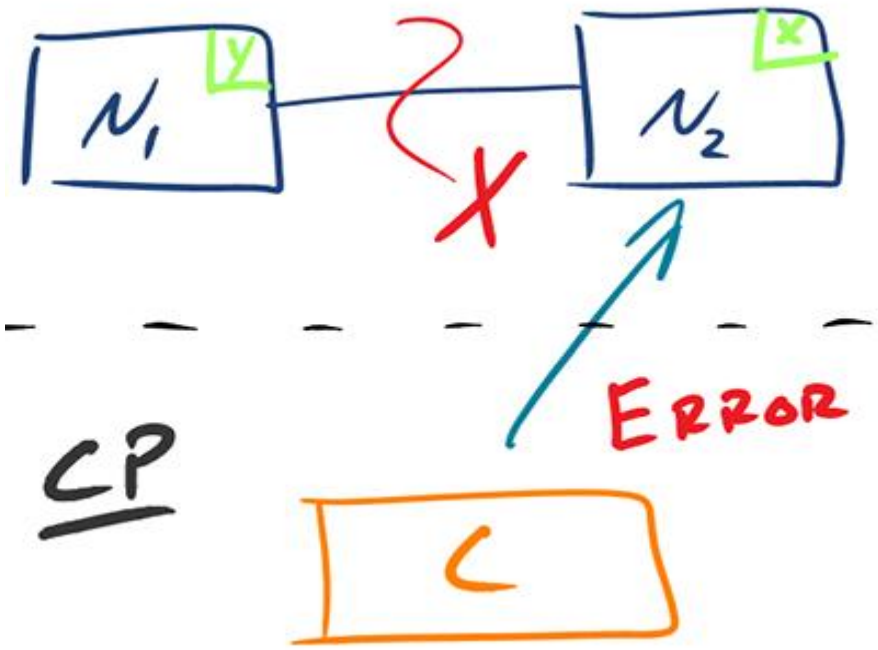
Consistency-Availability Tradeoff

We live in a physical world and can't guarantee the stability of a network, so distributed databases must choose Partition Tolerance (P). This implies a tradeoff between Consistency (C) and Availability (A).

CP - Consistency/Partition Tolerance

Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire.

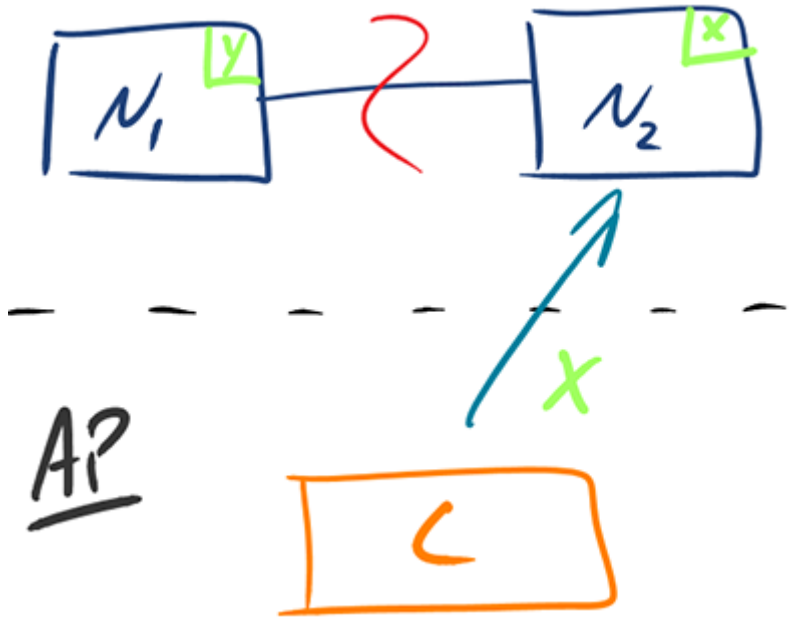
Choose Consistency over Availability when your business requirements dictate atomic reads and write



AP - Availability/Partition Tolerance

Return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved.

Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes. Availability is also a compelling option when the system needs to continue to function in spite of external errors (shopping carts, etc.)



The decision between Consistency and Availability is a software trade off. You can choose what to do in the face of a network partition - the control is in your hands. Network outages, both temporary and permanent, are a fact of life and occur whether you want them to or not - this exists outside of your software.

Building distributed systems provide many advantages, but also adds complexity. Understanding the trade-offs available to you in the face of network errors, and choosing the right path is vital to the success of your application. Failing to get this right from the beginning could doom your application to failure before your first deployment.

CA database

A CA database delivers consistency and availability across all nodes. It can't do this if there is a partition between any two nodes in the system, and therefore can't deliver fault tolerance.

Example: PostgreSQL, MariaDB.

CP database

A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node until the partition is resolved.

Example: MongoDB, Apache HBase.

AP database

An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. When the partition is resolved, the AP databases typically re-syncs the nodes to repair all inconsistencies in the system.

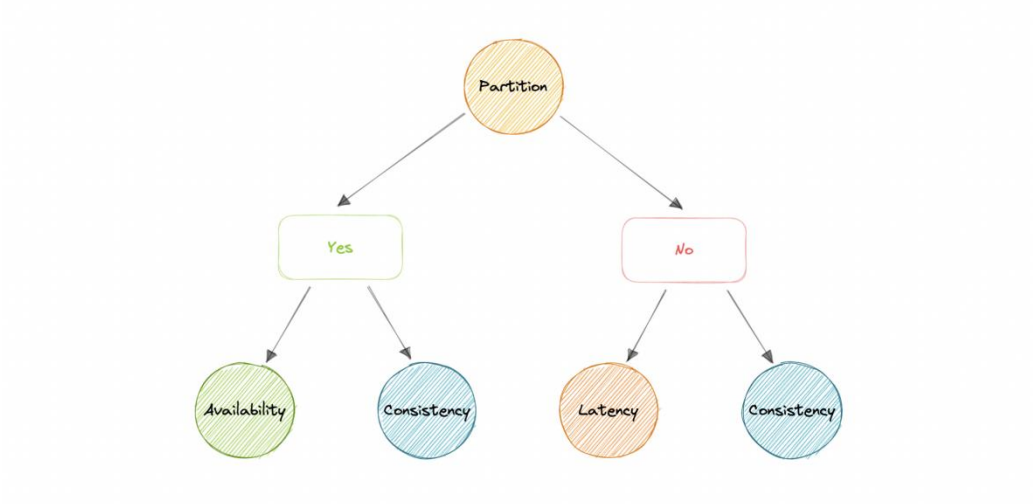
Example: Apache Cassandra, CouchDB.

PACELC Theorem

The PACELC theorem is an extension of the CAP theorem. The CAP theorem states that in the case of network partitioning (P) in a distributed system, one has to choose between Availability (A) and Consistency (C).

PACELC extends the CAP theorem by introducing latency (L) as an additional attribute of a distributed system. The theorem states that else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).

The PACELC theorem was first described by Daniel J. Abadi.



PACELC theorem was developed to address a key limitation of the CAP theorem as it makes no provision for performance or latency.

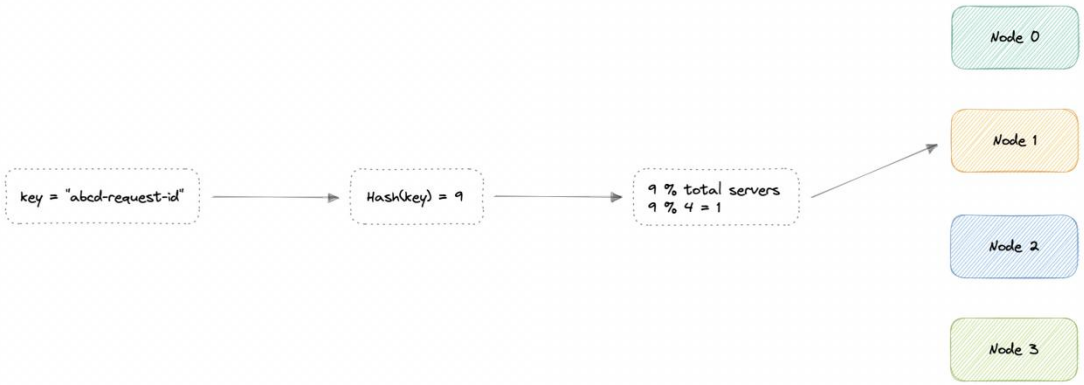
For example, according to the CAP theorem, a database can be considered available if a query returns a response after 30 days. Obviously, such latency would be unacceptable for any real-world application.

Consistent Hashing

Let's first understand the problem we're trying to solve.

Why do we need this?

In traditional hashing-based distribution methods, we use a hash function to hash our partition keys (i.e. request ID or IP). Then if we use the modulo against the total number of nodes (server or databases). This will give us the node where we want to route our request.



Hash(key1)→H1modN=Node0Hash(key2)→H2modN=Node1Hash(key3)→H3modN=Node2...Hash(keyn)→HnmodN=Noden−1

Where,

key: Request ID or IP.

H: Hash function result.

N: Total number of nodes.

Node: The node where the request will be routed.

The problem with this is if we add or remove a node, it will cause N to change, meaning our mapping strategy will break as the same requests will now map to a different server. As a consequence, the majority of requests will need to be redistributed which is very inefficient.

We want to uniformly distribute requests among different nodes such that we should be able to add or remove nodes with minimal effort. Hence, we need a distribution scheme that does not depend directly on the number of nodes (or servers), so that, when adding or removing nodes, the number of keys that need to be relocated is minimized.

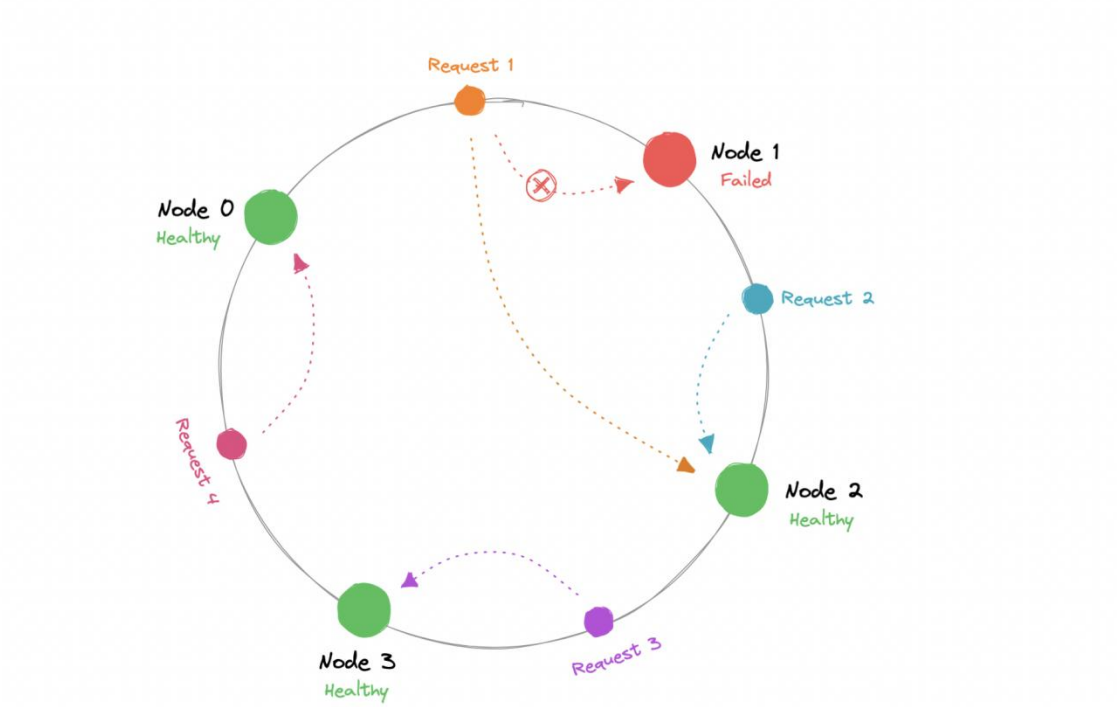
Consistent hashing solves this horizontal scalability problem by ensuring that every time we scale up or down, we do not have to re-arrange all the keys or touch all the servers.

Now that we understand the problem, let's discuss consistent hashing in detail.

How does it work

Consistent Hashing is a distributed hashing scheme that operates independently of the number of nodes in a distributed hash table by assigning them a position on an abstract circle, or hash ring.

This allows servers and objects to scale without affecting the overall system.



Using consistent hashing, only K/N data would require re-distributing.

$$R = K/N$$

Where,

R: Data that would require re-distribution.

K: Number of partition keys.

N: Number of nodes.

The output of the hash function is a range let's say 0...m-1 which we can represent on our hash ring. We hash the requests and distribute them on the ring depending on what the output was. Similarly, we also hash the node and distribute them on the same ring as well.

$$Hash(key1)=P1Hash(key2)=P2Hash(key3)=P3...Hash(keyn)=Pm-1$$

Where,

key: Request/Node ID or IP.

P: Position on the hash ring.

m: Total range of the hash ring.

Now, when the request comes in we can simply route it to the closest node in a clockwise (can be counterclockwise as well) manner.

This means that if a new node is added or removed, we can use the nearest node and only a fraction of the requests need to be re-routed.

In theory, consistent hashing should distribute the load evenly however it doesn't happen in practice. Usually, the load distribution is uneven and one server may end up handling the majority of the request becoming a hotspot, essentially a bottleneck for the system.

We can fix this by adding extra nodes but that can be expensive.

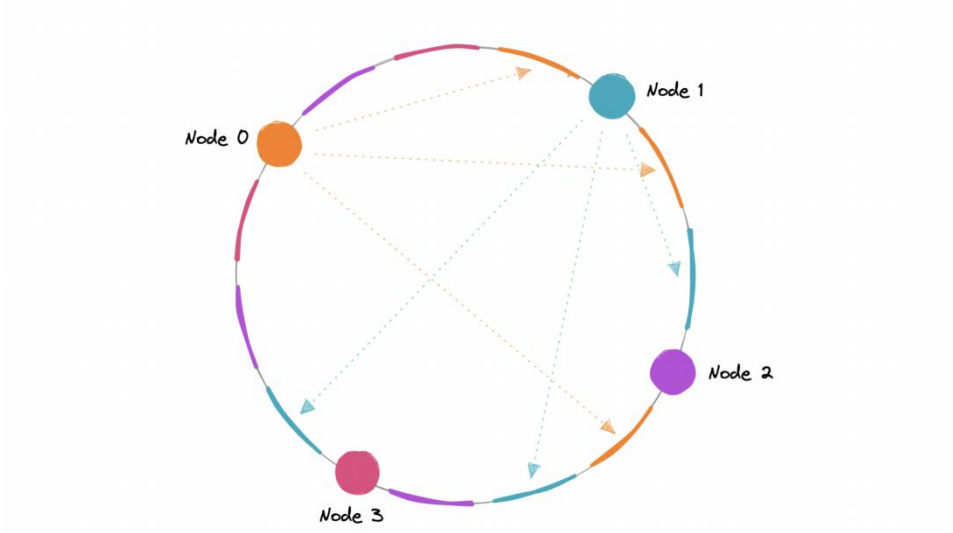
Let's see how we can address these issues.

Virtual Nodes

In order to ensure a more evenly distributed load, we can introduce the idea of a virtual node, sometimes also referred to as a V Node.

Instead of assigning a single position to a node, the hash range is divided into multiple smaller ranges, and each physical node is assigned several of these smaller ranges.

Each of these sub ranges is considered a V Node. Hence, virtual nodes are basically existing physical nodes mapped multiple times across the hash ring to minimize changes to a node's assigned range.



For this, we can use k number of hash functions.

Hash1(key1)=P1Hash2(key2)=P2Hash3(key3)=P3...Hashk(keyn)=Pm−1

Where,

key: Request/Node ID or IP.

k: Number of hash functions.

P: Position on the hash ring.

m: Total range of the hash ring.

As VNodes help spread the load more evenly across the physical nodes on the cluster by diving the hash ranges into smaller subranges, this speeds up the re-balancing process after adding or removing nodes. This also helps us reduce the probability of hotspots.

Data Replication

To ensure high availability and durability, consistent hashing replicates each data item on multiple N nodes in the system where the value N is equivalent to the replication factor.

The replication factor is the number of nodes that will receive the copy of the same data. In eventually consistent systems, this is done asynchronously.

Advantages

Let's look at some advantages of consistent hashing:

1. Makes rapid scaling up and down more predictable.
2. Facilitates partitioning and replication across nodes.
3. Enables scalability and availability.
4. Reduces hotspots.

Disadvantages

Below are some disadvantages of consistent hashing:

1. Increases complexity.
2. Cascading failures.
3. Load distribution can still be uneven.
4. Key management can be expensive when nodes transiently fail.

Examples

Let's look at some examples where consistent hashing is used:

1. Data partitioning in Apache Cassandra.
2. Load distribution across multiple storage hosts in Amazon DynamoDB.