Distributed Systems

GROUP 13

Technical Communication

Najma Baigum - 22114059
Preeti Karnwal -22114071
Shourya Goel - 22114090
Sukhman Singh - 22114097
Vangapandu Geethika - 22114104

Contents

Introduction of Distributed Systems

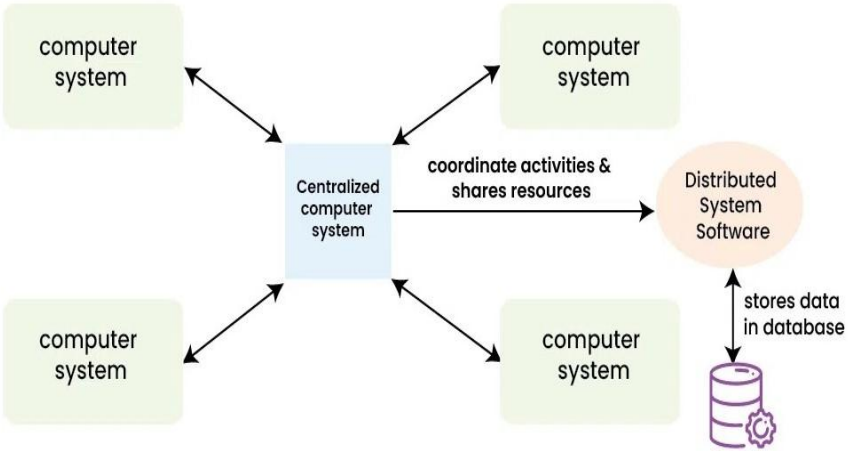
Distributed Systems at a high level

A **distributed system** is a collection of independent computers that work together as a single system, coordinating through network communication to achieve a shared goal. It provides scalability, fault tolerance, and resource sharing across multiple nodes.

There are two basic tasks that any computer system needs to accomplish:

- Storage and
- Computation

Upgrading hardware works for small problems, but as problems grow, hardware alone becomes insufficient or too costly. That's when we need distributed system to handle the load.



Applications

Cloud Computing

- On-demand access to computing resources, such as servers and storage

E-commerce Platforms

- Supports high scalability for handling massive concurrent user requests

Social Media Networks

- Manages large volumes of real-time interactions and content sharing.

Online Banking and Finance

- Ensures secure, real-time transactions and data redundancy.

IoT Networks

- Manages connected devices for data collection and remote control.

Scalability and other good things

It is the ability of a system , network or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

There are three particularly interesting things to look at:

Size scalability: adding more nodes should make the system linearly faster; growing the dataset should not increase latency.

Geographic scalability: allows multiple data centers to speed up responses while managing cross location delays.

Administrative scalability: adding nodes shouldn’t raise administrative costs or require more administrators per machine.

Performance

Performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

Depending on the context, this may involve achieving one or more of the following:

- Short response time/low latency for a given piece of work
- High throughput (rate of processing work)
- Low utilization of computing resource(s)

Latency

The state of being latent; delay, a period between the initiation of something and the occurrence.

In a distributed system, there is minimum latency that cannot be overcome: the speed of light limits how fast information can travel and hardware have a minimum latency cost incurred per operation..

Availability

The second aspect of a scalable system is availability.

It is the proportion of time a system is in a functioning condition. If a user cannot access the system, it is said to be unavailable.

Distributed systems help us achieve things that a single machine can't, like handling failures. The key is redundancy, which means having backups or duplicates. If one Part fails,others can keep the system running.Formulaically, availability is:

Availability = uptime/(uptime +downtime)

What prevents us from achieving good things?

Distributed systems are constrained by two physical factors:

- the number of nodes
- the distance between nodes

Working with constraints

- an increase in the number of independent nodes increases the probability of failure in a system
- an increase in the number of independent nodes may increase the need for communication between nodes.
- an increase in geographic distance increases the minimum latency for communication between distant nodes

Design techniques: partition and replicate

The manner in which a data set is distributed between multiple nodes is very important. In order for any computation to happen, we need to locate the data and then act on it.

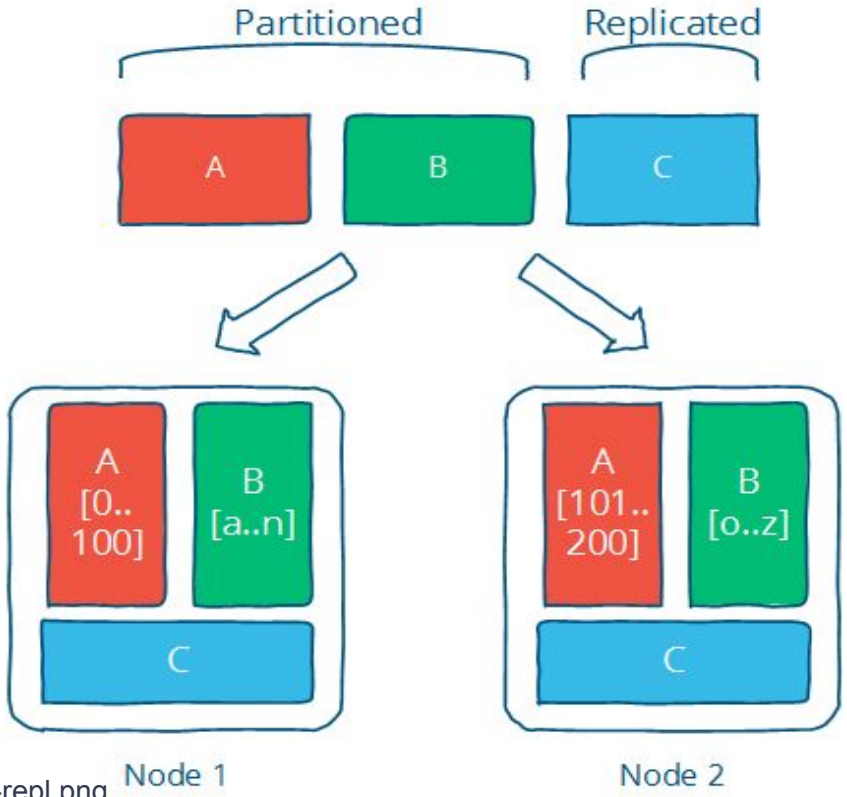
There are two basic techniques that can be applied to a data set:

- Partitioning
- Replication

Partitioning

Partitioning is dividing the dataset into smaller distinct independent sets.; this is used to reduce the impact of dataset growth. It also speed up queries by only checking relevant parts and grouping related data.

Partitioning improves availability by isolating failures, so other parts can keep running, making the system more reliable.



https://book.mixu.net/distsys/images/part-repl.png

Replication

Replication is making copies of the same data on multiple machines; this allows more servers to take part in the computation.

Replication - copying or reproducing something - is the primary way in which we can fight latency.

- Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data
- Replication improves availability by creating additional copies of the data, increasing the number of nodes that need to fail before availability is sacrificed

Replication is about providing extra bandwidth, and caching where it counts, maintaining consistency.

Replication allows us to achieve scalability, performance and fault tolerance.

Design elements in Distributed Systems

Synchronous & Asynchronous System Model

Synchronous System Model

- **Assumptions:**
  - Processes execute in lock-step, meaning there is a global clock and all processes have the same execution speed.
  - There is a known upper bound on message transmission delay.
  - Each process has a reliable clock with bounded accuracy.

Asynchronous System Model

- **Assumptions:**
  - No timing assumptions are made; processes can execute at different speeds.
  - There is no upper bound on message transmission delays (messages might take an arbitrarily long time to arrive).
  - Useful clocks do not exist, meaning there is no assumption that processes have synchronized clocks or that timing information can be relied upon.

The consensus problem

The consensus problem involves getting multiple nodes in a distributed system to agree on a single value. It is crucial for ensuring consistency and reliability in distributed systems.

Several computers (or nodes) achieve consensus if they all agree on some value. More formally:

1. Agreement: Every correct process must agree on the same value.
2. Integrity: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
3. Termination: All processes eventually reach a decision.
4. Validity: If all correct processes propose the same value V, then all correct processes decide V.

Impossibility Results

FLP Impossibility Result:

The FLP Impossibility result (Fischer, Lynch, and Patterson) addresses the consensus problem in the context of asynchronous systems. It is assumed that nodes can only fail by crashing; that the network is reliable, and that the typical timing assumptions of the asynchronous system model hold

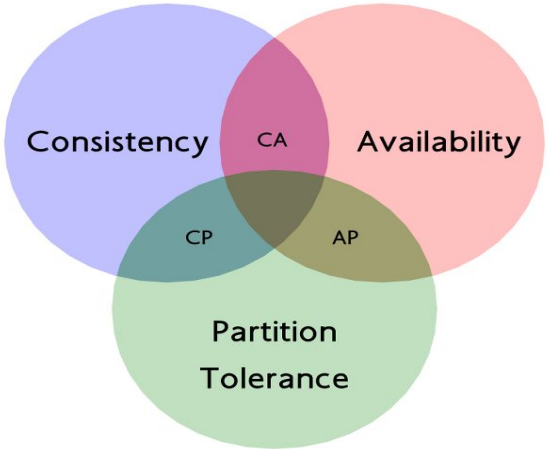
No deterministic algorithm can solve the consensus problem in an asynchronous system where nodes may crash, even with reliable message delivery and at most one failure.

An algorithm could remain undecided forever due to the lack of bounds on message delay, meaning it can't guarantee decision within a finite time.

CAP Theorem

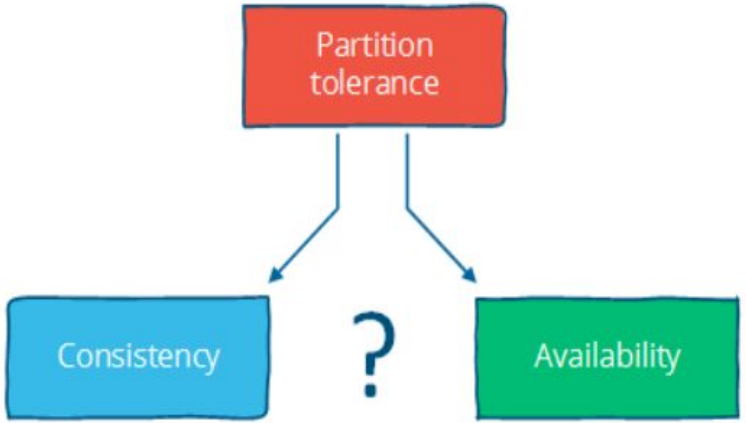
The theorem states that of these three properties:

- Consistency: all nodes see the same data at the same time.
- Availability: node failures do not prevent survivors from continuing to operate.
- Partition tolerance: the system continues to operate despite message loss due to network failure



Continued..

Assuming that a network partition occurs, the theorem reduces to a binary choice between availability and consistency.



Conclusions from CAP Theorem

- Many system designs used in early distributed relational database systems did not take into account partition tolerance
- There is a tension between strong consistency and high availability during network partitions.
- There is a tension between strong consistency and performance in normal operation.
- if we do not want to give up availability during a network partition, then we need to explore whether consistency models other than strong consistency are workable for our purposes.

Time and Order

Time and Order

Distributed programming is the art of solving the same problem that you can solve on a single computer using multiple computers.

Any system that can only do one thing at a time will create a total order of operations.

The easiest way to define "correctness" is to say "it works like it would on a single machine". And that usually means that a) we run the same operations and b) that we run them in the same order - even if there are multiple machines.

That is what makes distributed systems generic, they are not concerned with the instructions they are running but instead focus on preserving the order.

Total and Partial Order

Partial order (natural state in distributed systems) is defined as a state in which neither the network nor independent nodes make any guarantees about relative order; but at each node, you can observe a local order.

Total order (Ideal) is a binary relation that defines an order for every element in some set.

The former is realistic because there can exist some pairs of elements that are not comparable, think two branches in git diverging away from each other.

Both total order and partial order are transitive and antisymmetric.

`If a ≤ b and b ≤ a then a = b (antisymmetry);` `If a ≤ b and b ≤ c then a ≤ c (transitivity);`

However, a total order is total while a partial order is only reflexive:

`a ≤ b or b ≤ a (totality) for all a, b in X;` `a ≤ a (reflexivity) for all a in X;`

What is Time?

Time is a source of order - it allows us to define the order of operations

Timestamps really are a shorthand value for representing the state of the world from the start of the universe to the current moment. There are 3 interpretations :

1. Order : we can use timestamps to enforce a specific ordering of operations or the delivery of messages
2. Duration : durations measured in time have some relation to the real world.
3. Interpretation : time is a universally comparable value

By their nature, the components of distributed systems do not behave in a predictable manner

Does time progress at the same rate everywhere?

When implementing distributed systems we want to avoid making strong assumptions about time and order, because the stronger the assumptions, the more fragile a system is to issues with the "time sensor" - or the onboard clock.

The more temporal nondeterminism that we can tolerate, the more we can take advantage of distributed computation

Three answers to the above question :

1. Global Clock : Yes
2. Local Clock : Yes, but…
3. No Clock : No, duh :)

Time with a “Global-clock” assumption

The global clock assumption is that there is a global clock of perfect accuracy, and that everyone has access to that clock.
clock synchronization is limited by the lack of accuracy of clocks in commodity computers, by latency if a clock synchronization protocol such as NTP is used and fundamentally by the nature of spacetime.

Assuming that clocks on distributed nodes start at the same value and never drift apart is a nice assumption because you can use timestamps freely to determine a global total order - bound by clock drift rather than latency - but this is a nontrivial operational challenge and a potential source of anomalies.

Time with a “Local-clock” and “No-clock” assumption

A more plausible assumption is that each machine has its own clock, but there is no global clock ie. you cannot meaningfully compare timestamps from two different machines.

There is also the notion of logical time. Here, we don't use clocks at all and instead track causality in some other way like counters. This is a partial order as the events can be ordered on a single system using a counter and no communication, but ordering events across systems requires a message exchange.

When clocks are not used, the maximum precision at which events can be ordered across distant machines is bound by communication latency.

Vector Clocks (Time for causal order)

Lamport clocks and vector clocks are replacements for physical clocks which rely on counters and communication to determine the order of events across a distributed system. Lamport clock is defined as :

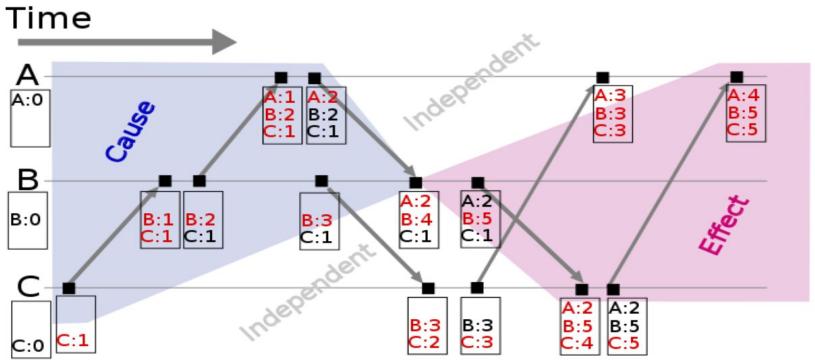
1. Whenever a process does work, increment the counter
2. Whenever a process sends a message, include the counter
3. When a message is received, set the counter to max(local_counter, received_counter) + 1

However, it defines a partial order, because a Lamport clock can only carry information about one timeline / history.

Hence comparing Lamport timestamps from systems that never communicate with each other may cause concurrent events to appear to be ordered when they are not

Continued..

A **vector clock** is an extension of Lamport clock, which maintains an array [t1, t2, ...] of N logical clocks - one per each node. Rather than incrementing a common counter, each node increments its own logical clock in the vector by one on each internal event.



The issue with vector clocks is mainly that they require one entry per node, which means that they can potentially become very large for large systems. A variety of techniques have been applied to reduce the size of vector clocks (either by performing periodic garbage collection, or by reducing accuracy by limiting the size).

Time, order and performance

While time and order are often discussed together, time itself is not such a useful property. We care about more abstract properties:

1. the causal ordering of events
2. failure detection (e.g. approximations of upper bounds on message delivery)
3. consistent snapshots (e.g. the ability to examine the state of a system at some point in time)

Now coming back to the genesis question, Is time / order / synchronicity really necessary? In some use cases, we want each intermediate operation to move the system from one consistent state to another. For example, a DBMS. But in other cases, we might not need that much synchronization. Say you are running a long computation, and don't really care about what the system does until the very end - then you don't really need much synchronization as long as you can guarantee that the answer is correct.

Replication in Single Copy Systems

Replication in Distributed Systems

In simple words, “Replication in distributed systems involves creating duplicate copies of data or services across multiple nodes”.

Replication causes many problems some of which being heavy storage requirements and synchronization problems, so why do we need it in distributed systems?

Replication allows us to achieve scalability, performance and fault tolerance. It allows more servers to take part in the computation.

High level design solutions of the replication problem

When we think of a distributed systems, some questions arise in the mind one of the most important of them being:

- What arrangement and communication pattern gives us the performance and availability characteristics we desire?

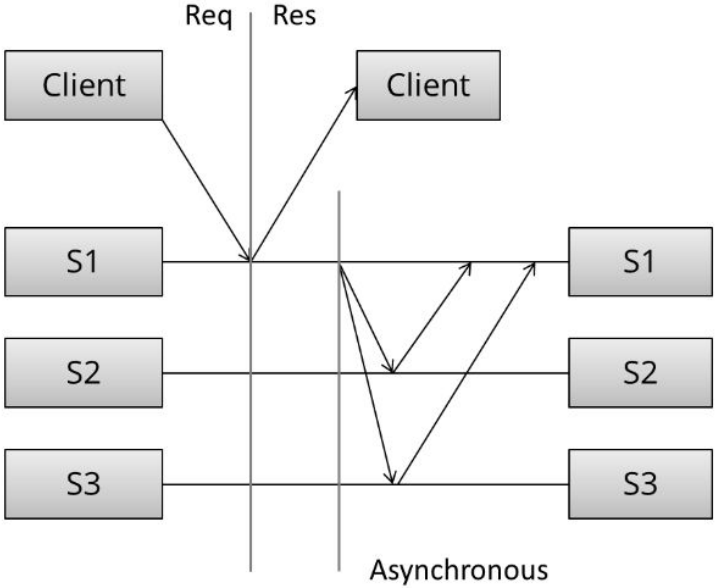
This led to the introduction of various algorithms each characterised by its own set of benefits and drawbacks.

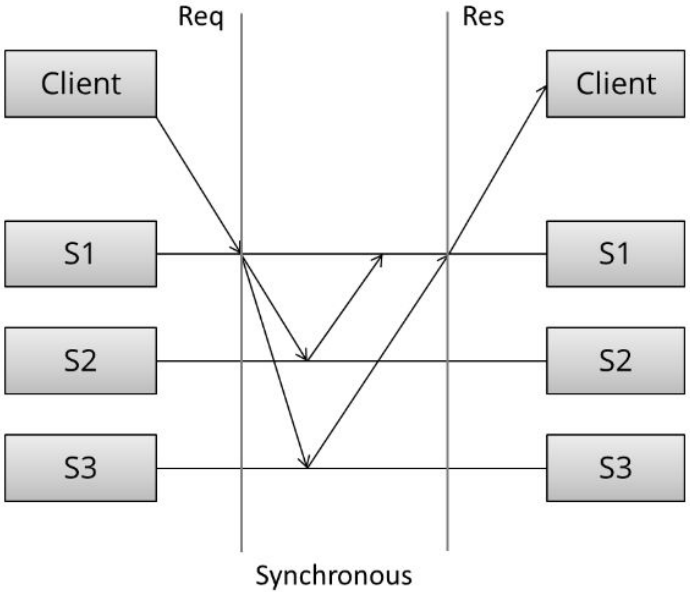
Synchronous and Asynchronous Replication

In **synchronous replication**, updates are committed to all replicas before acknowledging the write operation to the client. This ensures strong consistency but can introduce latency.
This is a write N - of - N approach: before a response is returned, it has to be seen and acknowledged by every server in the system.

In **asynchronous replication**, updates are propagated to replicas after the write operation is acknowledged to the client. This reduces latency but may lead to eventual consistency issues.
This is a write 1 - of - N approach: a response is returned immediately and update propagation occurs sometime later.

Continued..





Source: https://book.mixu.net/distsys/replication.html

Approaches to Replication

- Replication methods that prevent divergence (single copy systems) and
- Replication methods that risk divergence (multi-master systems)

The replication algorithms that maintain single-copy consistency include:

- 1n messages (asynchronous primary/backup)
- 2n messages (synchronous primary/backup)
- 4n messages (2-phase commit, Multi-Paxos)
- 6n messages (3-phase commit, Paxos with repeated leader election)

Primary/Backup Replication

Basic idea of primary-backup replication is to have two servers: the primary and the backup. We will design the protocol so that if either the primary or the backup crashes, clients will still be able to get a response from the system. It is the most commonly used replication method.

- Asynchronous P/B
- Synchronous P/B

Continued..

Any asynchronous replication algorithm can only provide weak durability guarantees which manifests as replication lag.

The synchronous variant of primary/backup replication ensures that writes have been stored on other nodes before returning back to the client - at the cost of waiting for responses from other replicas.

Two Phase Commit (2PC)

In the first phase (**voting**), the coordinator sends the update to all the participants. Each participant processes the update and votes whether to commit or abort.

In the second phase (**decision**), the coordinator decides the outcome and informs every participant about it.

```
[ Coordinator ] -> OK to commit?      [ Peers ]
                <- Yes / No

[ Coordinator ] -> Commit / Rollback  [ Peers ]
                <- ACK
```

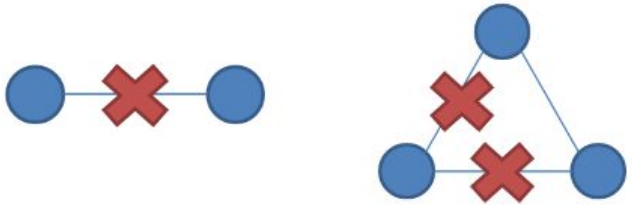
Source: https://book.mixu.net/distsys/replication.html

Continued..

The failure model that 2PC addresses does not include network partitions.

A network partition is the failure of a network link to one or several nodes. The nodes themselves continue to stay active, and they may even be able to receive requests from clients on their side of the network partition.

During a network partition, it is not possible to distinguish between a failed remote node and the node being unreachable.



Network Partition in (a) 2 nodes (b) 3 nodes

Source: https://book.mixu.net/distsys/replication.html

Partition Tolerant Consensus Algorithms

Partition tolerant consensus algorithms rely on a majority vote. As long as (N/2 + 1)-of-N nodes are up and accessible, the system can continue to operate.

When a network partition occurs, one partition will contain majority of nodes which will remain active and the partition with the minority nodes will stop processing to prevent divergence.

One example of these would be the Paxos Algorithm.

Features

Role: Consensus algorithms for replication generally opt for having distinct roles for each node.

Epochs: Each period of normal operation is known as an epoch. During each epoch only one node is the designated leader.

Duels: When a node detects that a leader has become non-responsive (or, in the initial case, that no leader exists), it switches to an intermediate state (called "candidate" in Raft) where it increments the term/epoch value by one, initiates a leader election and competes to become the new leader.

Replication in Multi Master Systems

Replication: weak consistency model protocols

A system enforcing strong consistency doesn't behave like a distributed system: it behaves like a single system, which is bad for availability during a partition.

Then, Why haven't weakly consistent systems been more popular?

Instead of having a single truth, we will allow different replicas to diverge from each other - both to keep things efficient but also to tolerate partitions - and then try to find a way to deal with the divergence in some manner.

Continued..

Eventual consistency with probabilistic guarantees: Conflicting updates will sometimes result in overwriting a newer value with an older one and some anomalies can be expected to occur during normal operation.

Eventual consistency with strong guarantees. : Such systems do not produce any anomalous results; without any coordination you can build replicas of the same service, and those replicas can communicate in any pattern and receive the updates in any order, and they will eventually agree on the end result as long as they all see the same information.

Continued..

CRDT's (convergent replicated data types) are data types that guarantee convergence to the same value in spite of network delays, partitions and message reordering.

The CALM (consistency as logical monotonicity) conjecture can be used to guide programmer decisions about when and where to use the coordination techniques from strongly consistent systems and when it is safe to execute without coordination.

Amazon Dynamo

Dynamo prioritizes availability over consistency.

Replicas may diverge from each other when values are written; when a key is read, there is a read reconciliation phase that attempts to reconcile differences between replicas before returning the value back to the client.A weakly consistent system can provide better performance and higher availability at a lower cost than a traditional RDBMS.

It uses **consistent hashing** for mapping keys to nodes to locate directly without querying the system.

Partial Quorums:

Dynamo uses partial quorums for reads and writes, allowing flexible trade-offs between consistency and availability. Users can specify:

- **W** (write quorum): The number of nodes required to acknowledge a write.
- **R** (read quorum): The number of nodes from which data is read.

A typical setting might be N=3 (3 replicas), with combinations like (R=2, W=2) or (R=1, W=2), which balance read speed and data durability.

Conflict Detection and Resolution:

- Dynamo allows replicas to diverge; conflicts are detected and resolved during reads using techniques like **vector clocks**. Clients may receive multiple conflicting versions of a value and are responsible for resolving them based on application-specific logic.
- Read repair techniques help to synchronize nodes gradually, improving the consistency of the data.

Replica Synchronization Using Gossip Protocol:

- Dynamo uses a **gossip protocol** for replica synchronization, ensuring data is eventually consistent across nodes. This decentralized approach scales well and avoids single points of failure.
- **Merkle trees** help efficiently identify differences between nodes, minimizing the data transferred during synchronization.
- It also maintains PBS which measures the consistency of read operations, balancing speed and data freshness.

Disorderly Programming and CRDTs

1. The Problem of Order-Dependence:

- In distributed systems, ensuring consistency across replicas without coordination is challenging.
- However, certain operations, such as reading and reconciling from multiple nodes or using commutative and associative operations, can achieve eventual consistency.

2. CRDTs: Convergent Replicated Data Types:

- CRDTs are data structures that ensure eventual consistency in distributed systems by exploiting operations that are commutative, associative, and idempotent. These properties ensure the result is the same regardless of operation order or duplication.
- CRDTs are structured around the mathematical concept of semilattices, which ensure convergence through a unique upper or lower bound.

The CALM theorem

CALM (Consistency as Logical Monotonicity) establishes a connection between logical monotonicity and eventual consistency. It posits that computations that are monotonic are inherently eventually consistent and can be safely executed without coordination.

Monotonic Logic vs. Non-Monotonic Logic:

- In monotonic logic, once a conclusion is reached based on certain premises, it remains valid even if more premises are added. This makes such computations retraction-free and incrementally more accurate.
- Non-monotonic logic allows conclusions to be invalidated by new information, which complicates consistency without coordination.

Conclusion

- **Scalability & Performance**: Distributed systems enable scalable and efficient data processing across multiple nodes, crucial for handling high demand and minimizing latency.
- **Fault Tolerance & Availability**: Through redundancy and replication, distributed systems achieve high availability and resilience against failures.
- **Design Trade-offs**: The CAP theorem highlights the balance between Consistency, Availability, and Partition tolerance, guiding system design based on application requirements.
- **Future Directions**: Emerging technologies like CRDTs and weak consistency models continue to enhance performance and manage data divergence.

Thank You