Distributed Systems in One Slide

Hailiang ZHAO @ ZJU-CS

http://hliangzhao.me

October 21, 2022

The slide summaries the key contents of an interesting online book *Distributed Systems for Fun and Profit*, assisted with several related materials.

Outline

Outline

Why Distributed Algorithms Important?

Distributed Programming

is the art of solving the same problem that you can solve on a single computer using multiple computers —— Usually, because the problem no longer fits on a single computer.

Ideally, adding a new machine would increase the performance and capacity of the system *linearly*. But of course this is not possible, because there is some overhead that arises due to having separate computers (data copy and partition cost, communication cost, etc.). This is why it's worthwhile to study distributed algorithms.

What We Want to Achieve

- **Scalability** 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.
  - size scalability, geographic scalability, and administrative scalability
- **Performance** is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.
  - response time
  - *latency*: the time during which something that has already happened is concealed from view
  - throughput
  - resource utilization
- **Availability** is the proportion of time a system is in a functioning condition.
  - *fault tolerance*: the ability of a system to behave in a well-defined manner once faults occur

Constraints on Distributed Systems

Distributed systems are constrained by two physical factors:

1. The number of nodes
2. The distance between nodes

Working within those constraints:

- When independent nodes increase,
  - the probability of failure increases
  - the communication between nodes increases
- When the geographic distance increases, the minimum latency for communication between distant nodes increases

Abstractions and Models

A good abstraction makes working with a system easier to understand, while capturing the factors that are relevant for a particular purpose. Typical models in distributed systems:

- System model (asynchronous / synchronous)
- Failure model (crash-fail, partitions, Byzantine)
- Consistency model (strong, eventual)

Partition and Replicate

The manner in which a data set is distributed between multiple nodes is very important.

- **Partitioning** is dividing the dataset into smaller distinct independent sets
  - Partitioning improves performance by limiting the amount of data to be examined and by locating related data in the same partition
  - Partitioning improves availability by allowing partitions to fail independently, increasing the number of nodes that need to fail before availability is sacrificed

Partition and Replicate

The manner in which a data set is distributed between multiple nodes is very important.

- **Replication** is making copies of the same data on multiple machines
  - 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 needs to follow a consistency model; Otherwise problems may appear. Only one consistency model for replication —— *strong consistency* —— allows you to program as-if the underlying data was not replicated

Outline

System Model Overview

System Model

is a set of assumptions about the environment and facilities on which a distributed system is implemented.

Typical assumptions include:

- what capabilities the *nodes* have and how they may fail
- how *communication links* operate and how they may fail
- properties of the overall system, such as assumptions about *time and order*

System Model on Nodes

Nodes serve as hosts for computation and storage. They have:

- the ability to execute a program
- the ability to store data into volatile memory (which can be lost upon failure) and into stable state (which can be read after a failure)
- a clock (which may or may not be assumed to be accurate)

The Crashing-Recovery Failure Model

is a model where nodes can only fail by crashing (stop executing), and can (possibly) recover after crashing at some later point.

The Byzantine Fault Tolerance Model

is a model where nodes can fail by misbehaving in any arbitrary way.

System Model on Communication Links

Communication links connect individual nodes to each other, and allow messages to be sent in either direction.

Network Partition

occurs when the network fails while the nodes themselves remain operational.

When network partition occurs, messages may be lost or delayed until the network partition is repaired.

Other rare assumptions: one-direction link, different communication costs for different link types, etc.

System Model on Time and Order

Timing assumptions arise because node experiences the world in a unique manner.

Synchronous System Model

Processes execute in lock-step; There is a known upper bound on message transmission delay; Each process has an accurate clock.

Asynchronous System Model

No timing assumptions, e.g., processes execute at independent rates; There is no bound on message transmission delay (A node cannot judge whether a message sent to it is *lost* or *delayed*); Useful clocks do not exist.

Real-world networks are subject to failures and there are no hard bounds on message delay.

The Consensus Problem

The Consensus Problem

Several processes must propose their candidate values, communicate with one another, and agree on a single consensus value. 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.

The FLP Impossibility Result

The FLP Impossibility Result

There DOES NOT EXIST a deterministic algorithm for the consensus problem in an asynchronous system subject to failures, even if

1. messages can never be lost (all messages are delivered correctly and exactly once),
2. at most one process may fail, and
3. it can only fail by crashing.

The CAP Theorem

The CAP Theorem

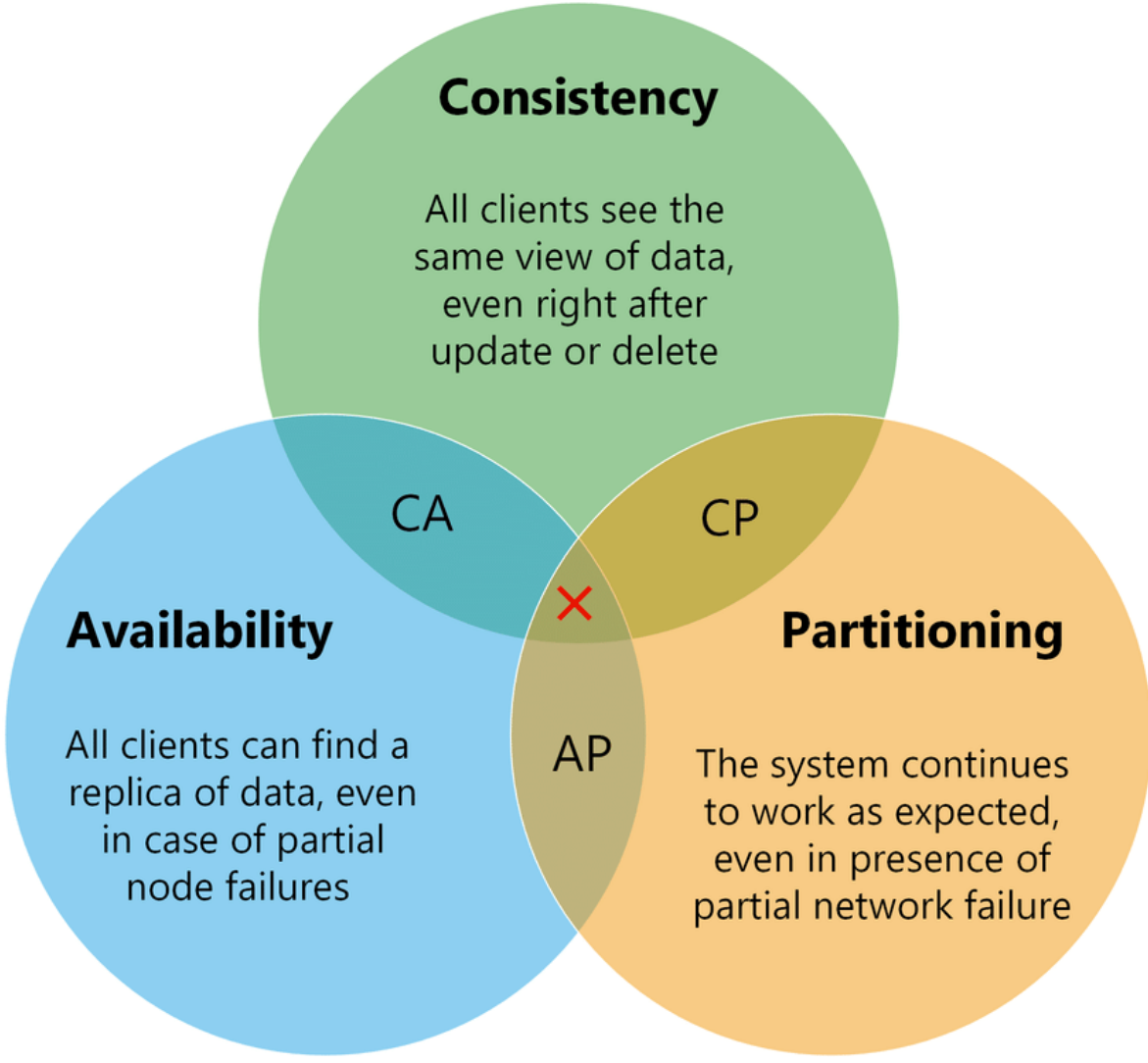
For the follow three properties:

- *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 and/or node failure

only two can satisfied simultaneously.

The CAP Theorem

We can get three different system types: CA, CP, and AP.



The CAP Theorem

We can get three different system types: CA, CP, and AP.

1. **CA**: Examples include full strict quorum protocols, such as *two-phase commit*.
2. **CP**: Examples include majority quorum protocols in which minority partitions are unavailable such as *Paxos*.
3. **AP**: Examples include protocols using conflict resolution, such as *Dynamo*.

The CAP Theorem

- WHY a CA system cannot be partition tolerance?
  A CA system does not distinguish between node failures and network failures, and hence must stop accepting writes everywhere to avoid introducing divergence (multiple copies). It cannot tell whether a remote node is down, or whether just the network connection is down: so the only safe thing is to stop accepting writes.
- WHY a CP system only supports single-copy consistency?
  A CP system prevents divergence (e.g. maintains single-copy consistency) by forcing asymmetric behavior on the two sides of the partition. It only keeps the majority partition around, and requires the minority partition to become unavailable (e.g. stop accepting writes), which retains a degree of availability (the majority partition) and still ensures single-copy consistency.

The CAP Theorem

What can we learn from the CAP theorm:

- Partition tolerance is an important property for modern systems and it SHOULD BE supported, since network partitions become much more likely, e.g., geographically distributed DCs.
- There is a tension between strong consistency and high availability during network partitions. This is because one CANNOT PREVENT divergence between two replicas that cannot communicate with each other while continuing to accept writes on both sides of the partition.
- 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.

More About Consistency

Consistency and availability are not really binary choices, unless you limit yourself to strong consistency. The "C" in CAP is "strong consistency", but "consistency" is not a synonym for "strong consistency".

A more detailed classification:

1. Strong consistency models (capable of maintaining a single copy)
   - 1.1 Linearizable consistency
   - 1.2 Sequential consistency
2. Weak consistency models
   - 2.1 Client-centric consistency models
   - 2.2 Causal consistency: strongest model available
   - 2.3 Eventual consistency models

Strong Consistency Models

Strong consistency models allow a programmer to replace a single server with a cluster of distributed nodes and not run into ANY problems.

- **Linearizable consistency**: Under linearizable consistency, all operations appear to be executed atomically in an order that is *consistent with the global real-time ordering of operations*.
- **Sequential consistency**: Under sequential consistency, all operations appear to have executed automatically in some order that is *consistent with the order seen at individual nodes and that is equal at all nodes* (allows for operations to be reordered as long as the order observed on each node remains consistent).

The only way to distinguish them is if they can observe all the inputs and timings going into the system; from the perspective of a client interacting with a node, they are *equivalent*.

Weak Consistency Models

- **Client-centric consistency models** are consistency models that involve the notion of a client or session in some way. For example, a client-centric consistency model might guarantee that a client will never see older versions of a data item (e.g., always fecth data from the addtional cache).
- **Eventual consistency model** says that if you stop changing values, then after some undefined amount of time all replicas will agree on the same value. Since it is trivially satisfiable (liveness property only), it is useless without supplemental information.

Outline

Total and Partial Order

A TOTAL ORDER is a binary relation that defines an order for *every* element in some set. Two distinct elements are **comparable** when one of them is greater than the other. In a PARTIALLY ORDERED set, some pairs of elements are not comparable and hence a partial order doesn't specify the exact order of every item.

- **Antisymmetry** (both Total and Partial):

$$\text{if } a \leq b \text{ and } b \leq a, \text{then } a = b, \forall a, b \in X. \tag{1}$$

- **Transitivity** (both Total and Partial):

$$\text{if } a \leq b \text{ and } b \leq c, \text{then } a \leq c, \forall a, b, c \in X. \tag{2}$$

Total and Partial Order

A TOTAL ORDER is a binary relation that defines an order for *every* element in some set. Two distinct elements are **comparable** when one of them is greater than the other. In a PARTIALLY ORDERED set, some pairs of elements are not comparable and hence a partial order doesn't specify the exact order of every item.

- **Total** (Total):

$$a \leq b \text{ or } b \leq a, \forall a, b \in X. \tag{3}$$

- **Reflexive** (Partial):

$$a \leq a, \forall a \in X. \tag{4}$$

Note that totality implies reflexivity; so a partial order is a weaker variant of total order.

Time and Timestamp

Time and timestamps have several useful interpretations when used in a program.

1. Order
2. Interpretation
3. Duration

By their nature, the components of distributed systems do not behave in a predictable manner. *They do not guarantee any specific order, rate of advance, or lack of delay.* Each node does have some local order —— as execution is (roughly) sequential —— but these local orders are independent of each other.

The synchronous system model has *a global clock*. The partially synchronous model has *a local clock*. In the asynchronous system model one cannot use clocks at all.

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.

1. The global clock is basically a source of total order (exact order of every operation on all nodes even if those nodes have never communicated)
2. Assuming that clocks on distributed nodes are perfectly synchronized means assuming that *clocks start at the same value and never drift apart*
3. We can use timestamps freely to determine *a global total order*

Time with a Local Clock Assumption

The local clock assumption assumes that each machine has its own clock, but there is no global clock.

1. We cannot use the local clock to determine whether a remote timestamp occurred before or after a local timestamp (timestamps from two machines are non-comparable)
2. It assigns a partial order: events on each system are ordered but events cannot be ordered across systems by only using a clock
3. We can use timestamps to order events on a single machine

Time with No Clock Assumption

With no clock assumption, we don't use clocks at all and instead track causality in some other way.

1. We can use *counters* and communication to determine whether something happened before, after or concurrently with something else.
2. With counters, we can determine the order of events between different machines, but cannot say anything about intervals and cannot use timeouts (since we assume that there is no "time sensor")
3. This is a partial order: events can be ordered on a single system using a counter and no communication, but ordering events across nodes requires a message exchange.

Use Time in Distributed Systems

The benefit of time:

1. *Time can define order across a system without communication*
2. *Time can define boundary conditions for algorithms*
   Timeout are widely used to determine whether a remote machine has failed, or whether it is simply experiencing high network latency

Lamport Clocks

Lamport clocks and vector clocks are replacements for physical clocks which rely on counters and communication to *determine the order of events.*

Lamport clock works as follows: Each process maintains a counter using the following rules:

- Whenever a process does work, increment the counter
- Whenever a process sends a message, include the counter
- When a message is received, set the counter to

$$\max\left(counter_{local}, counter_{received}\right) + 1. \tag{5}$$

Lamport Clocks

A Lamport clock allows counters to be compared across systems, with a caveat: Lamport clocks define a partial order. If $timestamp(a) < timestamp(b)$, then

- a may have happened before b
- a may be incomparable with b

If a and b are from the same causal history, e.g. either both timestamp values were produced on the same process; or b is a response to the message sent in a then we know that a happened before b.

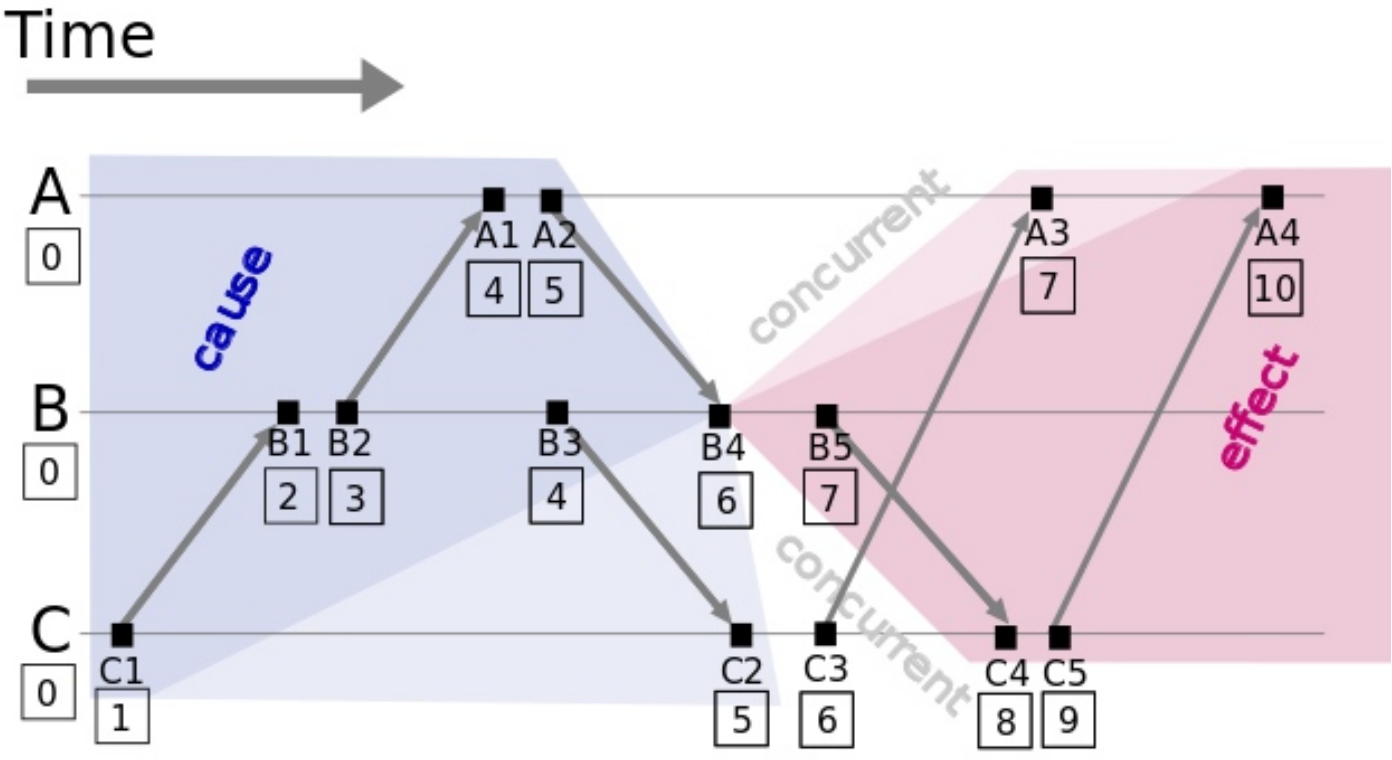
Vector Clocks

A vector clock is an extension of Lamport clock, which maintains an array $[t_1, ..., t_N]$ of N logical clcoks —— 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:

- Whenever a process does work, increment the logical clock value of the node in the vector
- Whenever a process sends a message, include the full vector of logical clocks
- When a message is received:
  - Update each element in the vector to be $\max(local, received)$
  - Increment the logical clock value representing the current node in the vector

Vector Clocks

An illustration on the vector clock:



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.

Failure Detectors

Given a program running on one node, how can it tell that a remote node has failed? In the absence of accurate information, we can infer that an *unresponsive* remote node has failed after some *reasonable amount of time* has passed.

A failure detector is a way to abstract away the exact timing assumptions. Failure detectors are implemented using *heartbeat messages and timers*. Failure detectors can be characterized by two properties, completeness and accuracy:

- Strong Completeness: Every crashed process is eventually suspected by *every* correct process
- Weak Completeness: Every crashed process is eventually suspected by *some* correct process
- Strong Accuracy: No correct process is suspected ever
- Weak Accuracy: Some correct process is never suspected

Failure Detectors

A failure detector with weak completeness can be transformed to one with strong completeness (by broadcasting information about suspected processes).

The difficulty lies in the incorrect suspection on correct process. If there is a hard maximum on the message delay (only exist in synchronous system model), strong accuracy can be achieved.

A better failure detector may output a suspicion level (a value $\in [0, 1]$) rather than a binary *up* or *down* judgement.

What We Really Care about When Discussing Time

While time and order are often discussed together, time itself is not such a useful property. Algorithms don't really care about time as much as they care about more abstract properties:

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

Outline

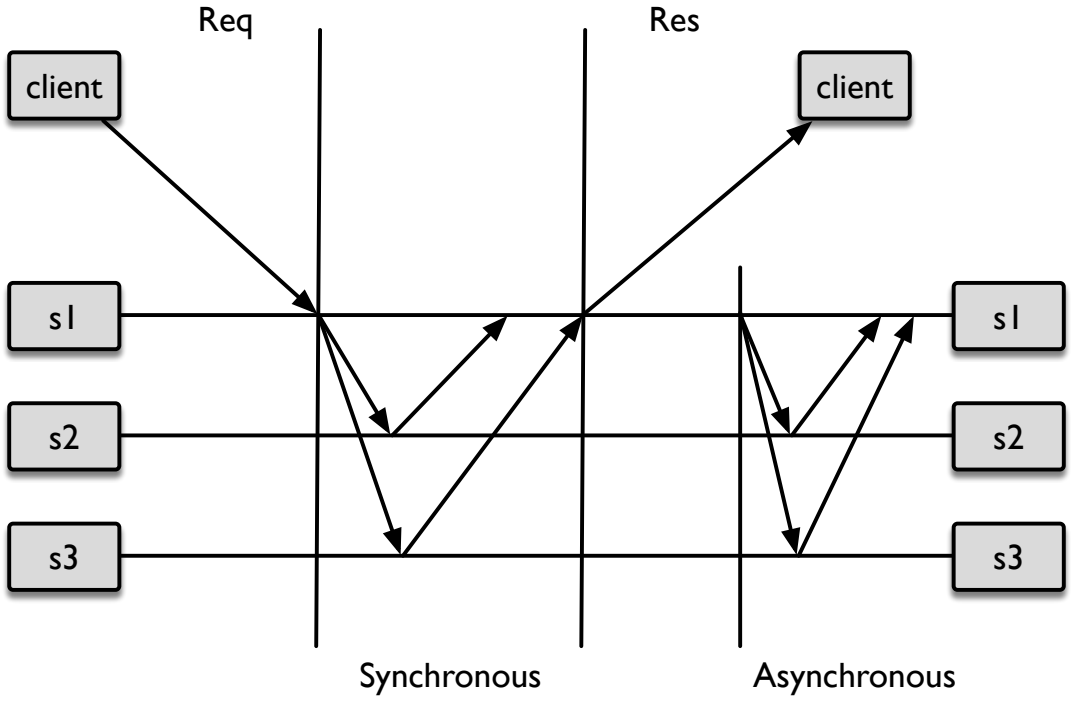
The Replication Problem

Replication is a group communication problem. For example,

- What arrangement and communication pattern gives us the performance and availability characteristics we desire?
- How can we ensure fault tolerance, durability and non-divergence in the face of network partitions and simultaneous node failure?

A General Communication Pattern

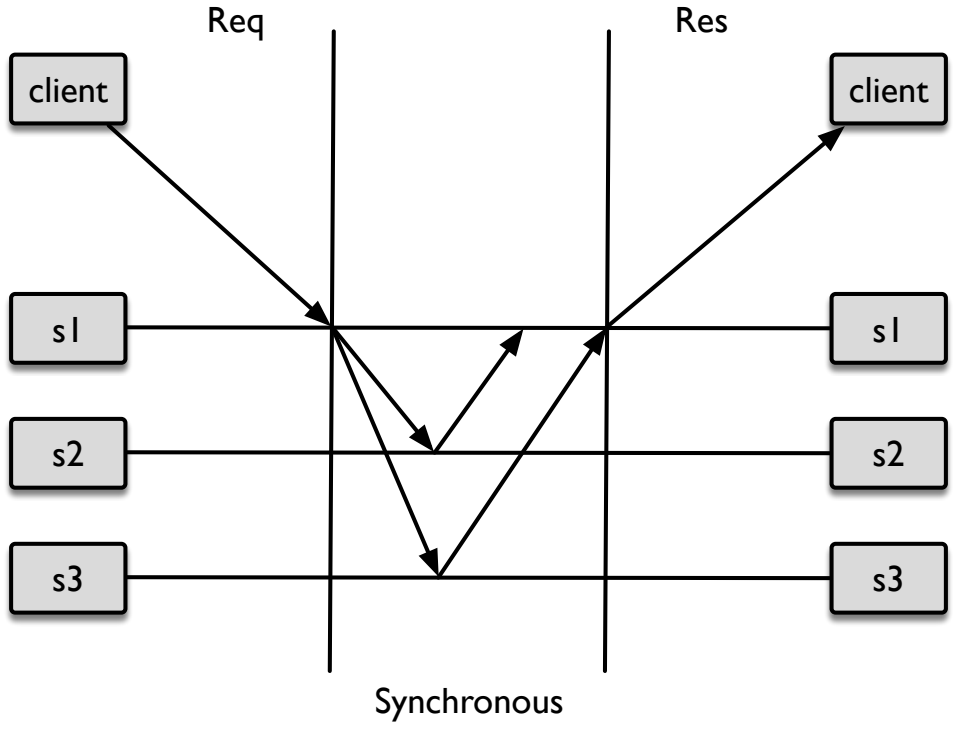
Assume that we have some initial database, and that clients make requests which change the state of the database.



1. (Request) The client sends a request to a server
2. (Sync) The synchronous portion of the replication takes place
3. (Response) A response is returned to the client
4. (Async) The asynchronous portion of the replication takes place

Synchronous Replication

Synchronous replication (a.k.a. active, or eager, or push, or pessimistic replication) looks like this:



During the synchronous phase, the first server contacts the two other servers and waits until it has received replies from all the other servers. Finally, it sends a response to the client informing it of the result (e.g. success or failure).

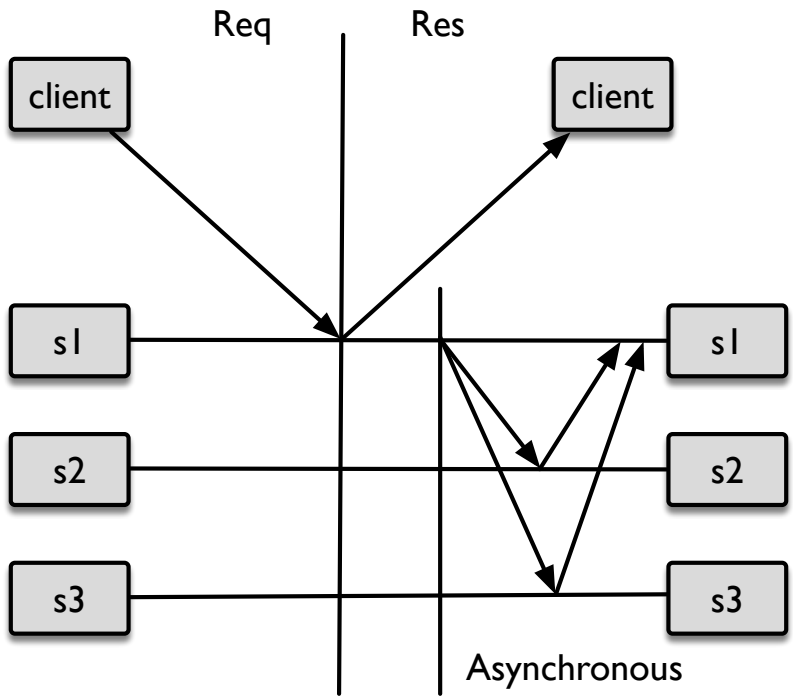
Synchronous Replication

Synchronous replication 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.

- The system cannot tolerate the loss of any servers
- Very strong durability guarantees are provided: The client can be certain that all N servers have received, stored and acknowledged the request when the response is returned

Asynchronous Replication

Asynchronous replication (a.k.a. passive replication, or pull replication, or lazy replication) looks like this:



The master/leader/coordinator immediately sends back a response to the client. It might at best store the update locally, but it will not do any significant work synchronously and the client is not forced to wait for more rounds of communication to occur between the servers.

Asynchronous Replication

Asynchronous replication is a write 1-of-N approach: A response is returned immediately and update propagation occurs sometime later.

- The system is fast and more tolerant of network latency
- This arrangement can only provide weak, or probabilistic durability guarantees: If nothing goes wrong, the data is eventually replicated to all N machines. However, if the only server containing the data is lost before this can take place, the data is permanently lost
- This arrangement cannot ensure that all nodes in the system always contain the same state

Major Replication Approaches

Except sync and async, replication techniques can also be divided into

- Replication methods that *prevent divergence* (single-copy systems): These methods *behave like a single system*
- Replication methods that *risk divergence* (multi-master systems)

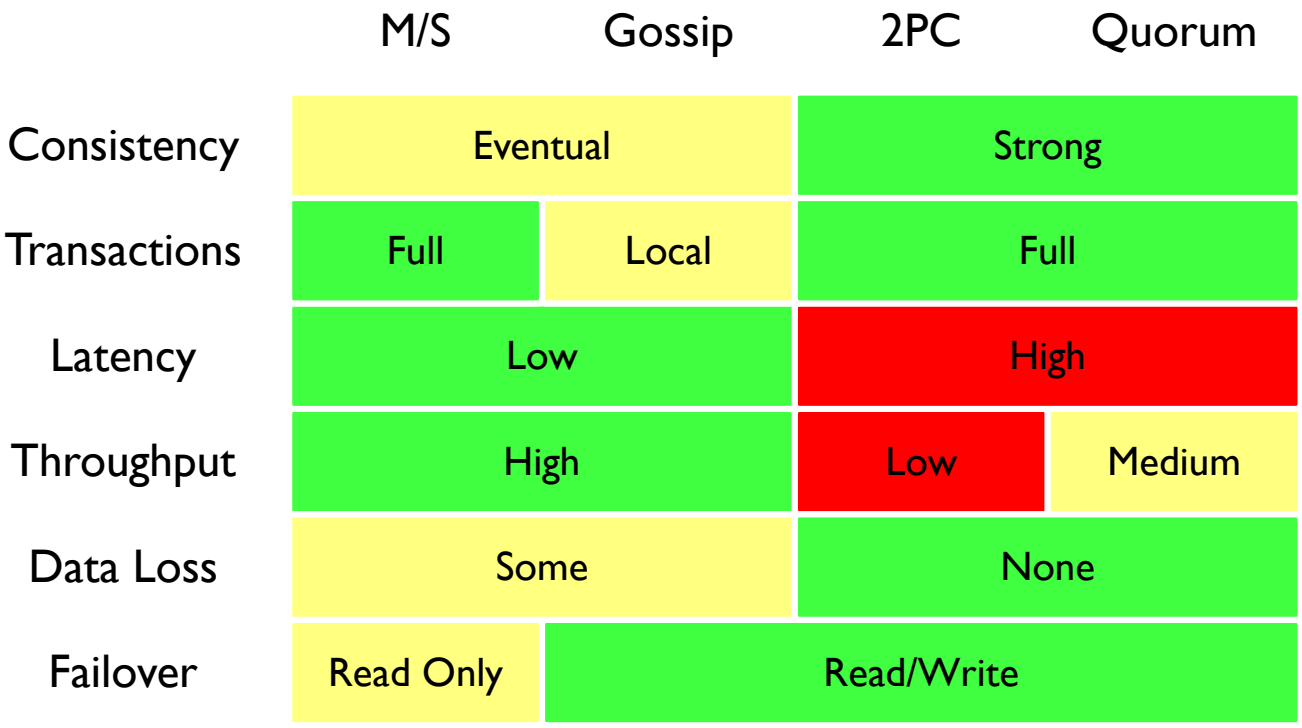
The replication algorithms that maintain single-copy consistency include:

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

More round of messages, more guarantees.

Major Replication Approaches

When you wait, you get worse performance but stronger guarantees. The following digram gives regular replication techniques and their properties.



Primary/Backup Replication

Primary/Backup Replication (P/B)

In P/B (a.k.a. primary copy replication/master-slave replication/log shipping), all updates are performed on the primary, and a log of operations (or alternatively, changes) is shipped across the network to the backup replicas.

1. asynchronous P/B: just *update*
2. synchronous P/B: *update* + *acknowledge receipt*

P/B can only offer a best-effort guarantee:

1. they are susceptible to lost updates or incorrect updates if nodes fail at inopportune times
2. they are susceptible to split-brain, where the failover to a backup kicks in due to a temporary network issue and causes both the primary and backup to be active at the same time (the *multi-master* problem)

Two-Phase Commit (2PC)

Adding another round of messaging to P/B, we get the Two-Phase Commit protocol (2PC).

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

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

- **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. When voting to commit, the participants store the update onto a temporary area (the write-ahead log). Until the second phase completes, the update is considered temporary

Two-Phase Commit (2PC)

Adding another round of messaging to P/B, we get the Two-Phase Commit protocol (2PC).

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

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

- **In the second phase (decision)**, the coordinator decides the outcome and informs every participant about it. If all participants voted to commit, then the update is taken from the temporary area and made permanent. Having the second pahse allows the system to *roll back* an update when a node fails

Two-Phase Commit (2PC)

Drawbacks of 2PC:

- 2PC is prone to blocking, since a single node failure (participant or coordinator) blocks progress until the node has recovered
- 2PC is a CA —— it is not partition tolerant. The failure model that 2PC addresses does not include network partitions; the prescribed way to recover from a node failure is to *wait until the network partition heals*

Network Partition

In the following several pages, we will discuss the common properties of partition-tolerant consensus algorithms, and introduce Paxos, Raft, and ZAB.

Network partition tolerance systems that enforce single-copy consistency requires that during a network partition, only ONE partition of the system remains active since during a network partition it is not possible to prevent divergence.

Typical partition tolerance consensus algorithms are Paxos and Raft.

Majority Decisions and Roles

Majority Votes

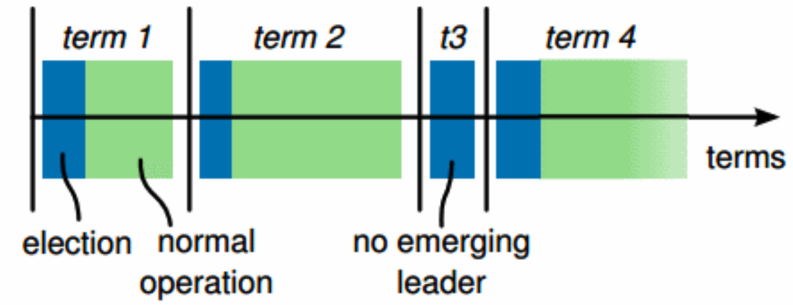
Partition tolerant consensus algorithms rely on **a majority vote**. As long as $(N/2 + 1)$-of-N nodes are up and accessible, the system can continue to operate.

Roles

Both Paxos and Raft make use of distinct node roles. In particular, they have a *leader* node (*proposer* in Paxos) that is responsible for coordination during normal operation. During normal operation, the rest of the nodes are *followers* (*acceptors* or *voters* in Paxos).

Epochs

Each period of normal operation in both Paxos and Raft is called an *epoch* (*term* in Raft).



- After a successful election, the same leader coordinates until the end of the epoch. As shown in the diagram above (from the Raft paper), some elections may fail, causing the epoch to end immediately.
- Epochs act as a logical clock, allowing other nodes to identify when an outdated node starts communicating —— nodes that were partitioned or out of operation will have a **smaller** epoch number than the current one, and their commands are ignored.

Leader Changes via Duels

All nodes start as followers; one node is elected to be a leader at the start. During normal operation, the leader maintains a **heartbeat** which allows the followers to detect if the leader *fails* or *becomes partitioned*.

From Candidate to Leader

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. To be elected a leader, a node must receive a majority of the votes.

From Candidate to Leader

One way to assign votes is to simply assign them on a **first-come-first-served** basis. In this way, a leader will eventually be elected.

Adding a random amount of waiting time between attempts at getting elected will reduce the number of nodes that are simultaneously attempting to get elected.

Numbered Proposals within an Epoch

During each epoch, the leader proposes one value at a time to be voted upon. **Within** each epoch, each proposal is numbered with a *unique* strictly increasing number. The followers (voters / acceptors) accept the first proposal they receive.

Proposals to be Submitted, Proposed, and Accepted

1. When a client submits a proposal (e.g. an update operation), the leader contacts all nodes in the quorum
2. If no competing proposals exist (based on the responses from the followers), the leader proposes the value
3. If a majority of the followers accept the value, then the value is considered to be accepted

Accepted Proposals' Number cannot be Changed

Since it is possible that another node is also attempting to act as a leader, we need to ensure that *once a single proposal has been accepted, its value can never change*. Otherwise a proposal that has already been accepted might be reverted by a competing leader.

Lamport states this as:

P2
If a proposal with value v has been chosen, then every higher-numbered proposal that is chosen has value v.

With this, both followers and proposers are constrained from changing a value that has been accepted by a majority.

Accepted Proposals' Number cannot be Changed

In order to enforce this property, the proposers must first ask the followers for their (highest numbered) accepted proposal and value. If the proposer finds out that a proposal already exists, then it must simply complete this execution of the protocol, rather than making its own proposal.

Lamport states this as:

P2b

If a proposal with value v is chosen, then every higher-numbered proposal issued by any proposer has value v.

The Core of Paxos

More specifically, we have:

P2c

For any value v and number n, if a proposal with v and n is issued (by a leader), then there is a set S consisting of a majority of acceptors (followers) such that either

1. *no follower in S has accepted any proposal numbered less than n, or*
2. *v is the value of the highest numbered proposal among all proposals numbered less than n accepted by the followers in S.*

This is the core of the Paxos algorithm. It means that, if multiple previous proposals exist, then the highest-numbered proposal value is proposed.

Paxos

With the above content, we give the two-round communication used in Paxos as follows:

```
[ Proposer ] -> Prepare(n)                                    [ Followers ]
             <- Promise(n; previous proposal number
                and previous value if accepted a
                proposal in the past)

[ Proposer ] -> AcceptRequest(n, own value or the value   [ Followers ]
                associated with the highest proposal number
                reported by the followers)
                <- Accepted(n, value)
```

- The prepare stage allows the proposer to learn of any competing or previous proposals
- The second phase is where either a new value or a previously accepted value is proposed

Paxos gives up liveness —— it may have to delay decisions indefinitely until a point in time where there are no competing leaders, and a majority of nodes accept a proposal.

ZAB and Raft

- ***ZAB*** *(Zookeeper Atomic Broadcast protocol)*
  Zookeeper is a system which provides coordination primitives for distributed systems, and is used by many Hadoop-centric distributed systems for coordination (e.g. HBase, Storm, Kafka). Zookeeper is basically the open source community's version of Chubby.
- ***Raft***
  It is designed to be easier to teach than Paxos, while providing the same guarantees. In particular, the different parts of the algorithm are more clearly separated and the paper also describes a mechanism for cluster membership change. It has recently seen adoption in etcd inspired by ZooKeeper.

Summary

In this section, we took a look at replication methods that enforce *strong consistency*.

- *P/B*
  - Single, static master
  - Replicated log, slaves are not involved in executing operations
    No bounds on replication delay
  - Not partition-tolerant
  - Manual/ad-hoc failover, not fault-tolerant
- *2PC*
  - Static master and unanimous vote: commit or abort
  - cannot survive from the failure of the coordinator and a node during a commit
  - Not partition-tolerant, tail latency-sensitive
- *Paxos*
  - Dynamic master and majority vote
  - Robust to $N/2 - 1$ simultaneous failures as part of protocol
  - Less sensitive to tail latency

Outline

Why We Care about Weak Consistency

Behaving like a single system by default is perhaps not desirable:

- 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
- For each operation, often a majority of the nodes must be contacted at least *twice*

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.

Thus we have eventual consistency —— *Nodes can for some time diverge from each other, but that eventually they will agree on the value.*

Two Designs for Eventual Consistency

1. **Eventual consistency with probabilistic guarantees**
   This type of system can detect conflicting writes at some later point, but does not guarantee that the results are equivalent to some correct sequential execution. In other words, 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 (or during partitions)
2. **Eventual consistency with strong guarantees**
   This type of system guarantees that the results converge to a common value equivalent to some correct sequential execution. In other words, such systems do not produce any anomalous results

Consistency as Logical Monotonicity

Some related terminologies:

CALM (Consistency as Logical Monotonicity)

If we can conclude that something is logically monotonic, then it is also safe to run without coordination.

CRDTs (Convergent Replicated Data Types)

CRDTs are data types that guarantee convergence to the same value in spite of network delays, partitions and message reordering.

Amazon's Dynamo

Dynamo is an eventually consistent, highly available key-value store.

K-V Store

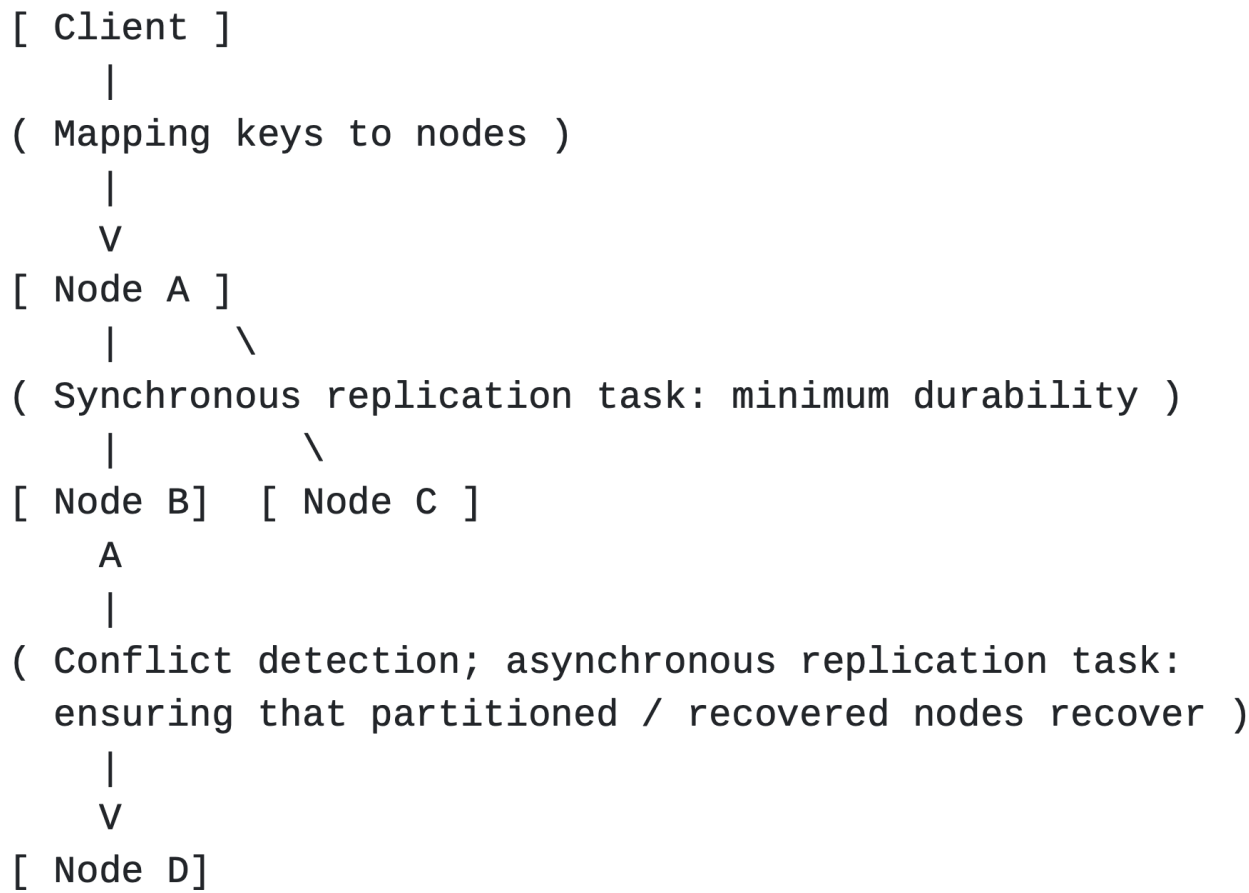
A K-V store is like a large hash table: a client can set values via $set(key, value)$ and retrieve them by key using $get(key)$.

A Dynamo cluster consists of N peer nodes; each node has a set of keys which is it responsible for storing. In Dynamo:

- 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

Amazon's Dynamo

The diagram below illustrates hwo Dynamo works. Specifically, how a **write** is routed to a node and written to multiple replicas.



Step 1: Mapping Keys to Nodes

As the digram above shows, whether we are reading or writing, the first thing that needs to happen is that we need to locate where the data should live on the system. This is done by ——

Consistent Hashing

With consistent hashing, a key can be mapped to a set of nodes responsible for it by a simple calculation on the client.

This means that a client can locate keys *without having to query the system* for the location of each key, which is much faster than RPCs.

Step 2: Synchronous Replication

Once we know where a key should be stored, we need to do some work to persist the value. This is a synchronous task; the reason why we will immediately write the value onto multiple nodes is *to provide a higher level of durability*.

Different from Paxos or Raft, Dynamo's quorums are **sloppy (partial) quorums** rather than strict (majority) quorums.

In partial quorums, different subsets of the quorum may contain different versions of the same data. The user can choose the number of nodes to write to and read from:

- W-of-N nodes required for a write
- W-of-N nodes required for a read

Step 2: Synchronous Replication

Writing to more nodes makes writes slower but increases the prob. that the value is not lost; reading from more nodes increases the prob. that the value read is up-to-date.

A typical configuration is $N = 3$ (e.g. a total of three replicas for each value). And, the usual recommendation is that

$$W + R > N, \tag{6}$$

because this means that the read and write quorums overlap in one node - making it less likely that a stale value is returned.

$W + R > N$ is not Equal to Strong Consistency

A system where $R + W > N$ can detect read/write conflicts, since any read quorum and any write quorum share a member. It guarantees that a previous write will be seen by a subsequent read.

However, this only holds if the nodes in N never change —— However, in Dynamo, the cluster membership can change if nodes fail.

Even $R = W = N$ would not guarantee strong consistency, since while the quorum sizes are equal to N, the nodes in those quorums can change during a failure.

Step 3: Conflict Detection and Read Repair

Systems that allow replicas to diverge must have a way to eventually reconcile two different values. In Dynamo, the conflict resolution is done by *tracking the causal history of a piece of data* by supplementing it with some metadata —— Clients must keep the metadata when they read, and must return back the metadata when writing.

Several ways to detect conflicts:

- *Timestamps*: The value with the higher timestamp value wins
- *Version numbers*
- *Vector clocks*: Concurrent and out of date updates can be detected. Performing read repair then becomes possible, though in some cases (concurrent changes) we need to ask the client to pick a value

Read Repair when Using Vector Clock

When reading a value, the client contacts R of N nodes and asks them for the *latest* value for a key. It takes *all* the responses, discards the values that are *strictly older* (using the vector clock value to detect this). And:

1. If there is only one unique vector clock + value pair, it returns that
2. If there are multiple vector clock + value pairs that have been edited concurrently (e.g. are not comparable), then all of those values are returned

Thus, the client must occasionally handle these cases by picking a value based on some use-case specific criterion.

Note that in a practical system the vector clock is not allowed to grow forever. Thus, garbage colleciton is required.

Step 4: Replication Synchronizaiton

Given that the Dynamo system design is tolerant of *node failures and network partitions*, it needs a way to deal with nodes rejoining the cluster after being partitioned, or when a failed node is replaced or partially recovered —— Replica synchronization is used to bring nodes up to date after a failure, and for periodically synchronizing replicas with each other. Specific techniques:

- *Gossip*: Nodes have some probability p of attempting to synchronize with each other. Every t seconds, each node picks a node to communicate with. This provides an additional mechanism beyond the synchronous task (e.g. the partial quorum writes) which brings the replicas up to date.

Step 4: Replication Synchronizaiton

Given that the Dynamo system design is tolerant of *node failures and network partitions*, it needs a way to deal with nodes rejoining the cluster after being partitioned, or when a failed node is replaced or partially recovered —— Replica synchronization is used to bring nodes up to date after a failure, and for periodically synchronizing replicas with each other. Specific techniques:

- *Merkle Trees*: A data store can be hashed at multiple different levels of granularity: a hash representing the whole content, half the keys, a quarter of the keys and so on. By maintaining this fairly granular hashing, nodes can compare their data store content much more efficiently than a naive technique. Once the nodes have identified which keys have different values, they exchange the necessary information to bring the replicas up to date.

Probabilistically Bounded Staleness (PBS)

The steps in Dynamo are:

1. consistent hashing to determine key placement
2. partial quorums for reading and writing
3. conflict detection and read repair via vector clocks
4. gossip for replica synchronization

The behavior of such a system can be characterized as Probabilistically Bounded Staleness (PBS).

PBS

PBS estimates the degree of inconsistency by using information about the anti-entropy (gossip) rate, the network latency and local processing delay to estimate the expected level of consistency of reads.

Convergent Replicated Data Types (CRDT)

Operations on CRDT is able to converge on the same value in an environment where replicas only communicate occasionally, the operations need to be order-independent and insensitive to (message) duplication/redelivery. The operations need to be:

- *Associative*: $a + (b + c) = (a + b) + c$
- *Commutative*: $a + b = b + a$
- *Idempotent*: $a + a = a$, so that duplication does not matter

Typical CRDTs

- **Counters**
  - *Grow-only counter* ($merge = \max(values)$; payload = single int)
  - *Positive-negative counter* (consists of two grow counters, one for increments and another for decrements)
- **Registers**
  - *Last-write-wins register* (timestamps or version numbers; $merge = \max(ts)$; payload = blob)
  - *Multi-valued register* (vector clocks; merge = take both)
- **Sets**
  - *Grow-only set* ($merge = \text{union}(items)$; payload = set; no removal)
  - *Two-phase set* (consists of two sets, one for adding, and another for removing; elements can be added once and removed once)
  - *Unique set* (an optimized version of the two-phase set)
  - *Last-write-wins set* ($merge = \max(ts)$; payload = set)
  - *Positive-negative set* (consists of one PN-counter per set item)
  - *observed-remove set*
- **Graphs and text sequences**

The CALM Theorem

Order-independence is an important property of any computation that *converges* —— If the order in which data items are received influences the result of the computation, then there is no way to execute a computation without guaranteeing order.

The CALM Theorem

The CLAM theorm states that logically monotonic programs are guaranteed to be eventually consistent.