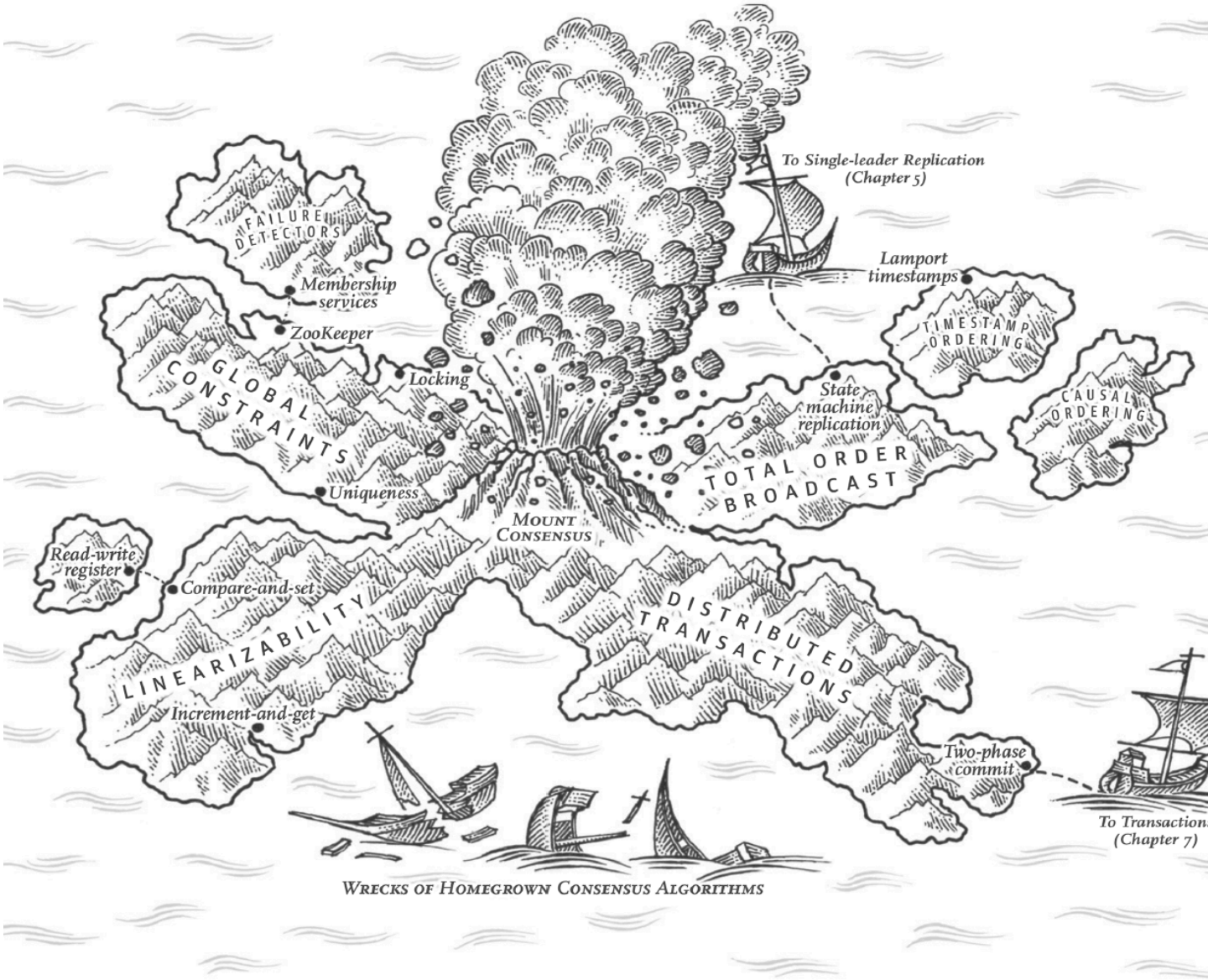

Chapter 9 Consistency and Consensus

We will talk about examples of algorithms and protocols for building fault-tolerant distributed systems.
The best way of building fault-tolerant systems is to find some general-purpose abstractions with useful guarantees, implement them once, and then let applications rely on the guarantees.
One of the most important abstractions for distributed systems is **consensus**: getting all of the nodes to agree on something.

Consistency Guarantees

Most replicated databases provide at least eventual consistency(week guarantee).
A better name for eventual consistency may be **convergence**, as we expect all replicas to eventually converge to the same value.
In this chapter we will explore stronger consistency models that data systems may choose to provide. Systems with stronger guarantees may have worse performance or be less fault-tolerant than systems with weaker guarantees.

Linearizability

- Linearizability / atomic consistency / strong consistency / immediate consistency / external consistency
- The basic idea is to make a system appear as if there were only one copy of the data, and all operations are atomic.
- With this guarantee, even though there may be multiple replicas, the application does not need to worry about them.
- In a linearizable system, when one client completes a write, all clients reading from the database must see the new value.
- Maintaining the illusion of a single copy of the data means guaranteeing that the value read is the most recent, up-to-date value.
- Linearizability is a recency guarantee.

What Makes a System Linearizable?

- If one client's read returns the new value, all following reads (on the same or other clients) must also return the new value, even if the write operation has not yet completed.

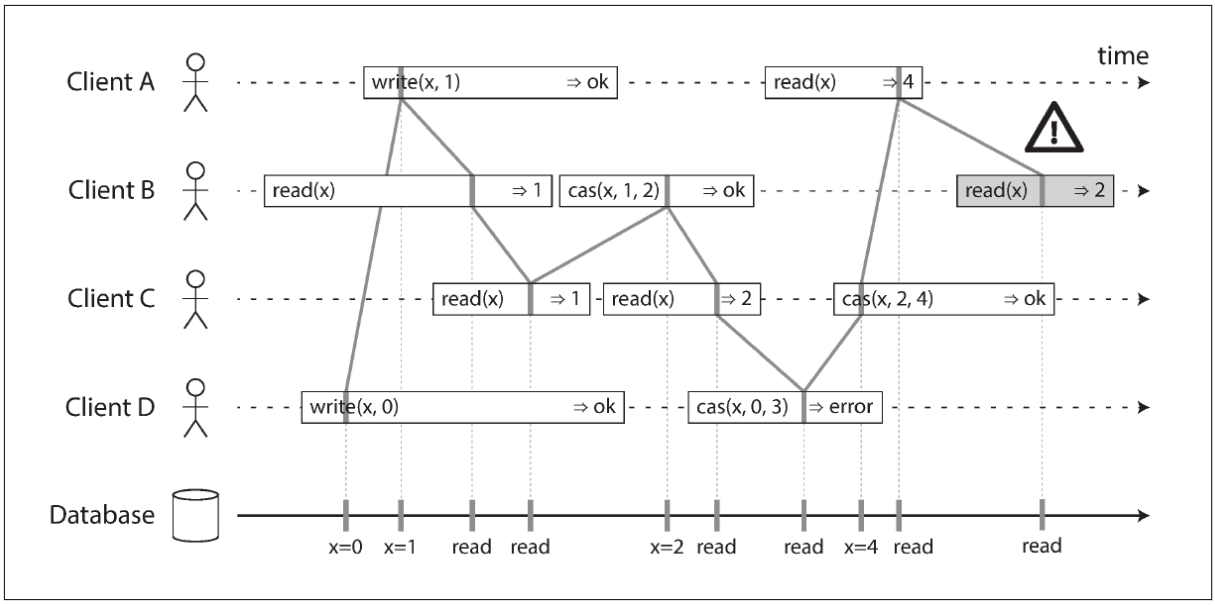

Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

Linearizability vs Serializability

- Both words seem to mean something like “can be arranged in a sequential order.”
- Two different guarantees:
    - **Serializability** is an **isolation property of transactions**(may read/write multiple objects). It guarantees that transactions behave the same as if they had executed in some serial order (each transaction running to completion before the next transaction starts). It is okay for that serial order to be different from the order in which transactions were actually run.
    - **Linearizability** is a **recency guarantee on reads and writes of a register**(an individual object). It doesn’t group operations together into transactions, so it does not prevent problems such as write skew, unless you take additional measures such as materializing conflicts.
- A database may provide both serializability and linearizability, and this combination is known as **strict serializability** or **strong one-copy serializability (strong-1SR)**. Implementations of serializability based on two-phase locking or actual serial execution are typically linearizable.
- However, serializable snapshot isolation is not linearizable: by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

Relying on Linearizability

In what circumstances is linearizability useful? When linearizability is an important requirement for making a system work correctly?

Locking and leader election

- A system that uses **single-leader replication** needs to ensure there is only one leader(otherwise split brain). One way of electing a leader is to use a lock: every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader. No matter how this lock is implemented, it must be linearizable: all nodes must agree which node owns the lock.
- **Coordination services like Apache ZooKeeper** are often used to implement distributed locks and leader election. They use consensus algorithms to implement linearizable operations in a fault-tolerant way.

Constraints and uniqueness guarantees

- Uniqueness constraints like a username or email address must uniquely identify per user, and in a file storage service there cannot be two files with the same path and filename.
- This situation is actually similar to a lock: when a user registers for your service, they acquire a “lock” on their chosen username. The operation is also very similar to an atomic compare-and-set.
- Similar constraints require a single up-to-date value (the account balance, the stock level, the seat occupancy) that all nodes agree on.
- A hard uniqueness constraint in relational databases requires linearizability. Other kinds of constraints (foreign key or attribute constraints) can be implemented without requiring linearizability.

Cross-channel timing dependencies

- The linearizability violation was only noticed because there was an additional communication channel in the system.

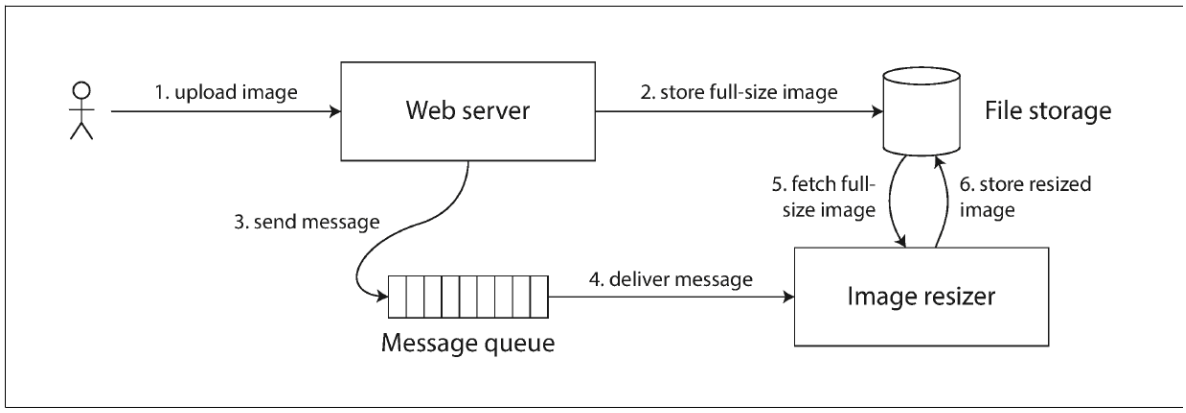

Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

- If the file storage service is not linearizable, there is the risk of a race condition: if the message queue is faster than the internal replication inside the storage service. When the resizer fetches the image, it might see an old version or nothing.

- This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue.
- Without the recency guarantee of linearizability, race conditions between these two channels are possible.
- Linearizability is not the only way of avoiding this race condition, but it's the simplest to understand. If you control the additional communication channel you can use alternative approaches similar to Reading Your Own Writes at the cost of additional complexity.

Implementing Linearizable Systems

The most common approach to making a system fault-tolerant is to use replication:

- Single-leader replication (potentially linearizable)
- Consensus algorithms (linearizable)
- Multi-leader replication (not linearizable)
- Leaderless replication (probably not linearizable)

Linearizability and quorums

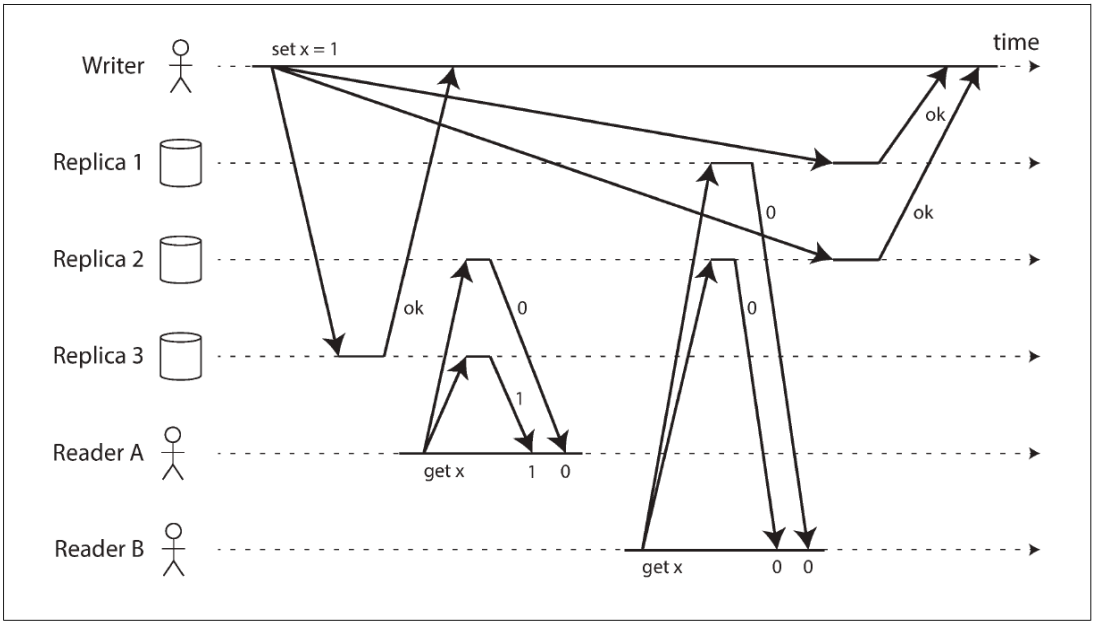

Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

- In the Dynamo-style model, when we have variable network delays, it is possible to have race conditions.
- The quorum condition is met (w + r > n), but this execution is nevertheless not linearizable.
- It is possible to make Dynamo-style quorums linearizable at the cost of reduced performance: a reader must perform read repair synchronously, before returning results to the application, and a writer must read the latest state of a quorum of nodes before sending its writes.

- Cassandra waits for read repair to complete on quorum reads, but it loses linearizability if there are multiple concurrent writes to the same key, due to its use of last-write-wins conflict resolution.
- Only linearizable read and write operations can be implemented this way; a linearizable compare-and-set operation cannot, because it requires a consensus algorithm.

The Cost of Linearizability

The CAP theorem

This issue is not just a consequence of single-leader and multi-leader replication: any linearizable database has this problem, no matter how it is implemented. The issue also isn't specific to multi-datacenter deployments, but can occur on any unreliable network, even within one datacenter. The trade-off:

- Application requires linearizability: some replicas are disconnected from the other replicas due to a network problem, they cannot process requests: must either wait until the network problem is fixed or return an error (become unavailable).
- Application does not require linearizability: each replica process requests independently, even if it is disconnected from other replicas (e.g., multi-leader). The application can remain available with behavior not linearizable.

CAP theorem: Consistency, Availability, Partition tolerance, pick 2 out of 3.

- At times when the network is working correctly, a system can provide both consistency (linearizability) and total availability.
- When a network fault occurs, you have to choose between either linearizability or total availability.
- CAP would be: either Consistent or Available when Partitioned.
- It only considers one consistency model (namely linearizability) and one kind of fault (network partitions, or nodes alive but disconnected from each other). It doesn't say anything about network delays, dead nodes, or other trade-offs.

Linearizability and network delays

- Although linearizability is a useful guarantee, few systems are actually linearizable in practice. E.g., RAM on a modern multi-core CPU is not linearizable.
- The reason for dropping linearizability is **performance**, not fault tolerance.

Ordering Guarantees

Ordering and Causality

- One of the reasons why ordering comes up so frequently is it helps preserve **causality**.
- **Causality** / **causal dependency** / **happened before relationship**
- If a system obeys the ordering imposed by causality, it is **causally consistent**.

- **Snapshot isolation provides causal consistency**: when you read from the database, and you see some piece of data, then you must also be able to see any data that causally precedes it (assuming it has not been deleted in the meantime).

The causal order is not a total order

A total order allows any two elements to be compared, you can tell which one is greater and which one is smaller.
Partially ordered: in some cases one set is greater than another, but in other cases they are **incomparable**.

The difference between a total order and a partial order is reflected in different database consistency models:

- **Linearizability:** In a linearizable system, we have a **total order** of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first.
- **Causality**: defines a **partial order**. Two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. Some operations are ordered with respect to each other, but some are incomparable.

Distributed version control systems as Git, their version histories like the graph of causal dependencies. Often one commit happens after another, in a straight line, but sometimes you get branches (when several people concurrently work on a project), and merges are created when those concurrently created commits are combined.

Linearizability is stronger than causal consistency

Relationship between the causal order and linearizability: **linearizability implies causality**, any system that is linearizable will preserve causality correctly.
Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remains available in the face of network failures.

Capturing causal dependencies

- Leaderless datastore needs to detect concurrent writes to the same key in order to prevent lost updates. Causal consistency goes further: it needs to track causal dependencies across the entire database, not just for a single key. Version vectors can be generalized to do this.
- In order to determine the causal ordering, the database needs to know which version of the data was read by the application.
  A similar idea is in the conflict detection of SSI(Serializable Snapshot Isolation). When a transaction wants to commit, the database checks whether the version of the data it read is up to date. The database keeps track of which data has been read by which transaction.

Sequence Number Ordering

- Actually keeping track of all causal dependencies is impractical. Application clients read lots of data before writing something, and then it is not clear whether the write is causally dependent on all or some of reads.
- A better way is to use **sequence numbers** or **timestamps** to order events. A timestamp need not come from a time-of-day clock, instead **a logical clock**, an algorithm to generate a sequence of numbers to identify operations, typically using **counters** that are incremented for every operation.
- Every operation has a **unique sequence number** thus providing a **total order**.
- We can create sequence numbers in a total order that is **consistent with causality**.
- In database with single-leader replication, the replication log defines a total order of write operations that is consistent with causality.

Noncausal sequence number generators

For multi-leader or leaderless database or partitioned database, methods to **generate sequence numbers** for operations:

1. Each node generates its own independent set of sequence numbers. Reserve some bits in the binary representation of the sequence number to contain a unique node identifier, this ensures two nodes generate different sequence numbers.
2. Attach a timestamp from a time-of-day clock (physical clock) to each operation. Used in the last write wins conflict resolution method.
3. Preallocate blocks of sequence numbers. Each node independently assigns sequence numbers from its block, allocates a new block when its supply of sequence numbers begins to run low.

They all have a problem: the sequence numbers they generate are not consistent with causality. Because these sequence number generators do not correctly capture the ordering of operations across different nodes:

1. Cannot accurately tell which one causally happened first on different nodes.
2. Timestamps from physical clocks are subject to clock skew, leading inconsistent with causality.
3. In the case of the block allocator, if one operation is given a sequence number from 1,001 to 2,000, and a causally later operation is given from 1 to 1,000. The sequence number is inconsistent with causality.

Lamport timestamps

A simple method to generate sequence numbers is consistent with causality: Lamport timestamps.

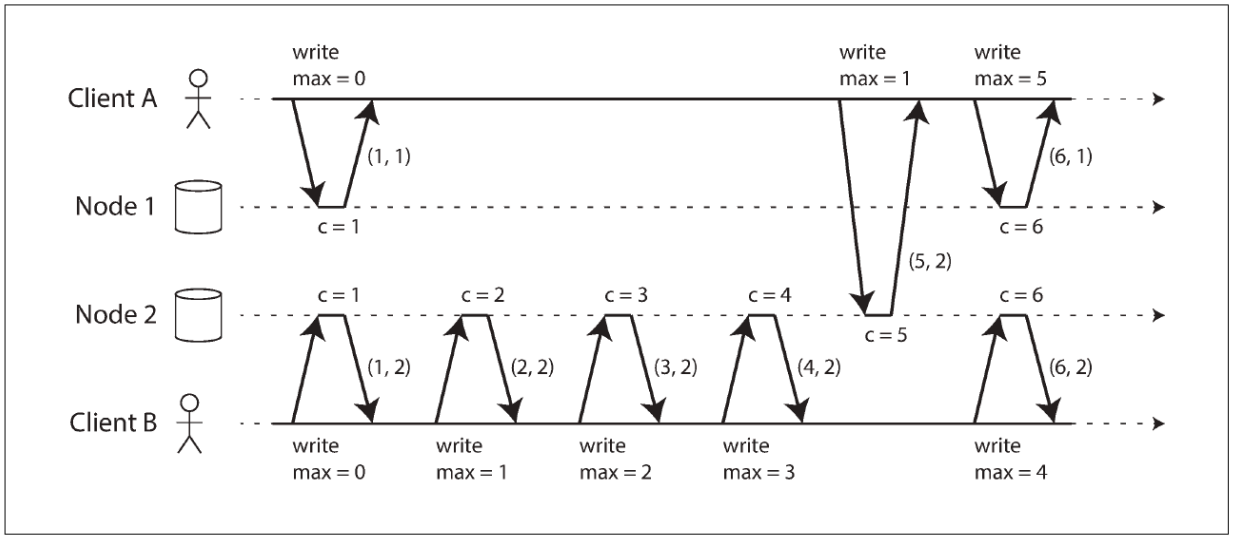

Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

- Each node has a unique identifier, each node keeps a counter of the number of operations it has processed.
- **The Lamport timestamp is a pair of (counter, node ID).**
- Two nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique.
- Lamport timestamp provides **total ordering**: if you have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp.

The key idea Lamport timestamps making consistent with causality is the following:

- Every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request.
- When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.
- Figure 9-8, where client A receives a counter value of 5 from node 2, and then sends that maximum of 5 to node 1. At that time, node 1's counter was only 1, but it was immediately moved forward to 5, so the next operation had an incremented counter value of 6.

Lamport timestamps are confused with version vectors:

They have a different purpose: version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas Lamport timestamps always enforce a total ordering. From the total ordering of Lamport timestamps, you cannot tell whether two operations are concurrent or whether they are causally dependent. The advantage of Lamport timestamps over version vectors is that they are more compact.

Timestamp ordering is not sufficient

To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations—you also need to know when that order is

finalized. If you have an operation to create a username, and you are sure that no other node can insert a claim for the same username ahead of your operation in the total order, then you can safely declare the operation successful.
This idea of knowing when your total order is finalized is captured in the topic of total order broadcast.

Total Order Broadcast

- Ordering by timestamps or sequence numbers is not as powerful as single-leader replication: if use timestamp ordering to implement a uniqueness constraint, it cannot tolerate any faults.
- Single-leader replication determines a total order of operations by choosing one node as the leader and sequencing all operations on a single CPU core on the leader.
- The challenge is how to scale the system if the throughput is greater than a single leader can handle, and how to handle failover if the leader fails. In the distributed systems, this problem is **total order broadcast** or **atomic broadcast**.

Scope of ordering guarantee: Partitioned databases with a single leader per partition often maintain ordering only per partition, they cannot offer consistency guarantees (e.g., consistent snapshots, foreign key references) across partitions. Total ordering across all partitions is possible with additional coordination.

Total order broadcast:
A protocol for exchanging messages between nodes. It requires two **safety properties**:

- **Reliable delivery**: No messages are lost: if a message is delivered to one node, it is delivered to all nodes.
- **Totally ordered delivery**: Messages are delivered to every node in the same order.

Using total order broadcast

- **Consensus services as ZooKeeper** implement total order broadcast.
- **Total order broadcast is what you need for database replication**: if every message represents a write to the database, and every replica processes the same writes in the same order, then the replicas remain consistent with each other (aside from any temporary replication lag). This principle is **state machine replication.**

- Total order broadcast can be used to implement **serializable transactions**. if every message represents a deterministic transaction to be executed as a stored procedure, and if every node processes messages in the same order, then the partitions and replicas of the database are kept consistent with each other.

- An important aspect of total order broadcast is that **the order is fixed at the time the messages are delivered**: a node is not allowed to retroactively insert a message into an earlier position in the order if subsequent messages have already been delivered. This fact makes **total order broadcast stronger than timestamp ordering**.

- Total order broadcast is useful for implementing a **lock service that provides fencing tokens**. Every request to acquire the lock is appended as a message to the log, and all messages are sequentially numbered in the order they appear in the log. The sequence number can then serve as a fencing token, because it is monotonically increasing. In **ZooKeeper**, this sequence number is called **zxid**.

Implementing linearizable storage using total order broadcast

Total order broadcast is asynchronous: messages are guaranteed to be delivered in a fixed order, but no guarantee about when a message will be delivered.
Linearizability is a recency guarantee: a read is guaranteed to see the latest value.

You can build linearizable storage on top of total order broadcast.

For every possible username, you have a linearizable register with an atomic compare-and-set operation. Every register initially has the value null (username is not taken). When a user wants to create a username, execute a compare-and-set operation on the register for username, setting it to the user account ID, the condition is the previous register value is null. If multiple users try to concurrently grab the same username, only one of the compare-and-set operations will succeed, because the others will see a value other than null(due to linearizability).

Implement such a **linearizable compare-and-set operation** as follows by using **total order broadcast** as an **append-only log**:

1. Append a message to the log, indicating the username you want to claim.
2. Read the log, wait for the message you appended to be delivered back to you.
3. Check for any messages claiming the username that you want.
   If the first message for your desired username is your own message, then you are successful, commit the username claim and acknowledge it to the client.
   If the first message for the desired username is from another user, abort the operation.

Log entries are delivered to all nodes in the same order, if there are concurrent writes, all nodes agree on which one came first. Choosing the first of the conflicting writes as the winner and aborting later ones ensures all nodes agree on whether a write was committed or aborted. A similar approach can be used to implement serializable multi-object transactions on top of a log.

This procedure only guaratees linearizable writes, not linearizable reads. Reads may be stale from a store asynchronously updated from the log. (The procedure provides **sequential consistency** or **timeline consistency**, a weaker guarantee than **linearizability**).
Few options to make reads linearizable:

- Sequence reads through the log by appending a message, reading the log, and performing the actual read when the message is delivered back. The message's position in the log defines the time at read happens.
- If the log allows you to fetch the position of the latest log message in a linearizable way, you can query that position, wait for all entries up to that position to be delivered to you, then perform the read. (This is the idea behind ZooKeeper's sync() operation)

- You can make your read from a replica that is synchronously updated on writes, and is thus sure to be up to date. (This is used in chain replication).

Implementing total order broadcast using linearizable storage

- The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation. Alternatively, an atomic compare-and-set operation also works.
- **Algorithm**: for every message sent through total order broadcast, you increment-and-get the linearizable integer, then attach the value from the register as a sequence number to the message. You can then send the message to all nodes (resending any lost messages), and the recipients will deliver the messages consecutively by sequence number.
- **Key difference between total order broadcast and timestamp ordering**: Unlike Lamport timestamps, the numbers you get from incrementing the linearizable register form **a sequence with no gaps**. Thus, if a node has delivered message 4 and receives an incoming message with a sequence number of 6, it knows that it must wait for message 5 before it can deliver message 6. This is not the case with Lamport timestamps.
- Make a linearizable integer with an atomic increment-and-get operation, the problem lies in handling the situation when network connections to that node are interrupted, and restoring the value when that node fails.
- Linearizable sequence number generators can end up with a consensus algorithm.
- A linearizable compare-and-set (or increment-and-get) register and total order broadcast are both equivalent to a consensus problem. If one of these problems is solved, you can transform into a solution for the others.

Distributed Transactions and Consensus

Consensus is one of the most important and fundamental problems in distributed computing. The goal is simply to get several nodes to agree on something.

Use cases:

- **Leader election**: Database with single-leader replication, all nodes need to agree on which node is the leader. Avoid bad failover, split brain, inconsistency and data loss.
- **Atomic commit**: Transactions in several nodes or partitions, all nodes to agree on the outcome of the transaction: either they all abort or commit.

Consensus algorithms:

- **Two-phase commit (2PC) algorithm** is the most common way of solving atomic commit, implemented in various databases, messaging systems, and application servers. But it is not a very good one.
- Better consensus algorithms, used in ZooKeeper (Zab) and etcd (Raft).

Atomic Commit and Two-Phase Commit (2PC)

Atomicity ensures the secondary index consistent with the primary data.

From single-node to distributed atomic commit

It is not enough to send a commit request to all of the nodes and independently commit the transaction on each one. The commit succeeds on some nodes and fails on other nodes, violating atomicity guarantee. (requests in some nodes have constraint violation or conflict, requests lost in the network, nodes crash)

Introduction to two-phase commit

Two-phase commit is an algorithm to achieve atomic transaction commit across multiple nodes. The commit/abort process in 2PC is split into two phases.

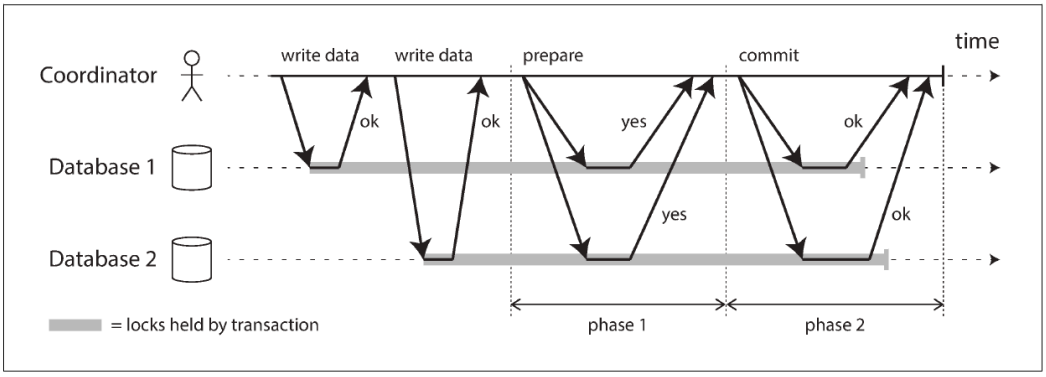

Figure 9-9. A successful execution of two-phase commit (2PC).

Don't confuse 2PC and 2PL:

- Two-phase commit (2PC): provides atomic commit in a distributed database.
- Two-phase locking (2PL): provides serializable isolation.

2PC uses a component: a **coordinator** / **transaction manager**. The coordinator is implemented as a library within the same application process that is requesting the transaction, and can also be a separate process or service.

Steps:

- A 2PC transaction begins with the application reading and writing data on multiple database (**participants**) nodes.
- When the application is ready to commit, the coordinator begins phase 1: it sends a prepare request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:
- If all participants reply "yes", the coordinator sends out a commit request in phase 2, and the commit actually takes place.
- If any of the participants replies "no," the coordinator sends an abort request to all nodes in phase 2.

This process is somewhat like the traditional marriage ceremony in Western cultures.

A system of promises

Break down the process in more detail:

1. When the application wants to begin a distributed transaction, it requests a **globally unique transaction ID** from the **coordinator**.
2. The application begins a **single-node transaction** on all **participants**, and attaches the **globally unique transaction ID** to the single-node transaction. All reads and writes are done in one of these single-node transactions. If anything goes wrong, the coordinator or any of the participants can abort.
3. When the application is ready to commit, the coordinator sends a **prepare request** to all participants, tagged with the **global transaction ID**. If any of these requests fails or times out, the coordinator sends an abort request for that transaction ID to all participants.
4. When a participant receives the prepare request, it makes sure that it can definitely commit the transaction under all circumstances. This includes writing all transaction data to disk (crash, power failure, or running out of disk space), and checking for any conflicts or constraint violations. By replying “yes” to the coordinator, the node promises to commit the transaction without error if requested, without actually committing it.
5. When the coordinator has received responses to all prepare requests, it makes a decision on whether to commit or abort the transaction. **The coordinator writes the decision to its transaction log on disk** so that it knows the decision in case it crashes. This is the **commit point**.
6. Once the coordinator’s decision has been written to disk, the commit or abort request is sent to all participants. **If this request fails or times out, the coordinator must retry forever until it succeeds**. If a participant has crashed in the meantime, the transaction will be committed when it recovers.

The protocol has two crucial “points of no return”: when a participant votes “yes,” it promises to definitely commit later; and once the coordinator decides, that decision is irrevocable. Those promises **ensure the atomicity of 2PC**. (Single-node atomic commit lumps these two events into one: writing the commit record to the transaction log.)
Same as the marriage analogy.

Coordinator failure

- If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction. But once the participant has voted “yes,” it must wait to hear back from the coordinator whether to commit or abort. If the coordinator crashes or the network fails at this point, the participant can do nothing but wait. A participant’s transaction in this state is called in **doubt** or **uncertain**.
- Without hearing from the coordinator, the participant has no way to know whether to commit or abort. In principle, the participants could communicate among themselves to find out how each participant voted and come to some agreement, but that is not part of the 2PC protocol.
- The only way 2PC can complete is by waiting for the coordinator to recover. The coordinator must write its commit or abort decision to a transaction log on disk before

sending commit or abort requests to participants: when the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log. Any transactions that don't have a commit record in the coordinator's log are aborted. Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.

Three-phase commit

- Two-phase commit is called a **blocking atomic commit protocol** due to the fact that 2PC can become stuck waiting for the coordinator to recover. Making atomic commit protocol nonblocking is not straightforward.
- Alternative algorithm three-phase commit (3PC): assumes a network with bounded delay and nodes with bounded response times; in most practical systems with unbounded network delay and process pauses, it cannot guarantee atomicity.
- Nonblocking atomic commit requires a perfect failure detector, a reliable mechanism for telling whether a node has crashed or not. In a network with unbounded delay a timeout is not a reliable failure detector, because a request may time out due to a network problem even if no node has crashed.

Distributed Transactions in Practice

Distributed transaction implementations have heavy performance penalties. Much performance cost in two-phase commit is due to the additional disk forcing (fsync), required for crash recovery, and additional network round-trips.

Two different types of distributed transactions:

- **Database-internal distributed transactions**: Distributed databases support internal transactions among the nodes of that database. No need to be compatible with any other system, they can use any protocol and apply optimizations specific to that particular technology.
- **Heterogeneous distributed transactions**: The participants are two or more different technologies. E.g., two databases from different vendors, or non database systems such as message brokers. A distributed transaction across these systems must ensure atomic commit, even the systems may be different under the hood.

Exactly-once message processing

Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways.

For example, a message from a message queue can be acknowledged as processed if the database transaction for processing the message was successfully committed. This is implemented by atomically committing the message acknowledgment and the database writes in a single transaction.

If either the message delivery or the database transaction fails, both are aborted, and the message broker can redeliver the message later.

By **atomically committing the message and the side effects of its processing**, we can ensure that the message is effectively processed exactly once.
All systems have to use the same atomic commit protocol. For example, say a side effect of processing a message is to send an email, and the email server does not support two-phase commit: it could happen that the email is sent two or more times if message processing fails and is retried.

XA transactions

- **X/Open XA** (short for **eXtended Architecture**) is **a standard for implementing two-phase commit across heterogeneous technologies.**
- Supported by **traditional relational databases** (PostgreSQL, MySQL, SQL Server, Oracle) and **message brokers**.
- Not a network protocol, it is an **API for interfacing** with a **transaction coordinator**. API bindings exist in other languages: In Java EE applications, XA transactions are implemented using the **Java Transaction API (JTA)**, which in turn is supported by many drivers for databases using **Java Database Connectivity (JDBC)** and drivers for message brokers using the **Java Message Service (JMS)** APIs.
- XA assumes that your application uses a **network driver** or **client library** to communicate with the **participant databases** or **messaging services**. If the driver supports XA, it calls the XA API to find out whether an operation should be part of a distributed transaction. If so, it sends information to the database server. The driver also exposes **callbacks** through which the coordinator can ask the participant to prepare, commit, or abort.
- **The transaction coordinator implements the XA API**. The coordinator is a **library** loaded into the same process as the application issuing the transaction (**not a separate service**). It keeps track of the participants in a transaction, collects partipants' responses after asking them to prepare (via a callback into the driver), and uses a log on the local disk to keep track of the commit/abort decision for each transaction.
- If the application process crashes, the coordinator goes with it. Any participants with prepared but uncommitted transactions are then stuck in doubt. Since the coordinator's log is on the application server's local disk, when the server restarted, the coordinator library read the log to recover the commit/abort of each transaction. The coordinator uses the database driver's XA callbacks to ask participants to commit or abort. The database server cannot contact the coordinator directly, since all communication must go via its client library.

Holding locks while in doubt

- Can the rest of the system ignore the in-doubt transaction?
  The problem is with **locking**. In "Read Committed", database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes. In serializable isolation, a database using two-phase locking also has to take a shared lock on any rows read by the transaction.
- The database cannot release locks until the transaction commits or aborts.

In a two-phase commit, a transaction must hold the locks throughout the time it is in doubt.
If the coordinator crashes and takes long to start up again, or coordinator's log is lost, locks will be held forever until manually resolved by an administrator.
This can cause large parts of your application unavailable until the in-doubt transaction is resolved.

Recovering from coordinator failure

- In theory, if the coordinator crashes and restarts, it should recover its state from the log and resolve any in-doubt transactions.
- In practice, orphaned in-doubt transactions do occur, transactions for which the coordinator cannot decide the outcome (e.g., transaction log lost or corrupted due to a software bug). These transactions cannot be resolved automatically, so they sit forever in the database, holding locks and blocking other transactions.
- Rebooting database servers will not fix this problem, since a correct implementation of 2PC must preserve the locks of an in-doubt transaction even across restarts (otherwise it violates the atomicity guarantee).
- The only way is an administrator to manually decide whether to commit or roll back the transactions. The administrator examines the participants of each in-doubt transaction, determines whether any participant has committed or aborted already, and applies the same outcome to the other participants. Resolving the problem requires a lot of manual effort, and needs to be done under high stress and time pressure during a serious production outage
- Many XA implementations have an **emergency escape hatch** called **heuristic decisions**: allowing a participant to unilaterally decide to abort or commit an in-doubt transaction without a definitive decision from the coordinator.
- Heuristic is for probably breaking atomicity, since it violates the system of promises in two-phase commit. Thus, heuristic decisions are intended only for getting out of catastrophic situations, and not for regular use.

Limitations of distributed transactions

XA transactions solve the real and important problem of keeping several participant data systems consistent with each other, they also introduce major operational problems.
Transaction coordinator is a kind of database (store transaction outcomes), it needs be treated as other database:

- If the coordinator is not replicated, it is a single point of failure for the entire system. Many coordinator implementations are not highly available by default.
- Many **server-side applications** are developed in a **stateless model (as favored by HTTP**), with all persistent state stored in a database, which has the advantage that application servers can be added and removed at will. However, when the coordinator is part of the application server, it changes the nature of the deployment.
  Coordinator's logs become a crucial part of the durable system state, as important as the databases themselves. **Such application servers are no longer stateless**.

- Since XA needs to be **compatible** with a wide range of data systems, it is necessarily a lowest common denominator. For example, it **cannot detect deadlocks across different systems** (since that require a standardized protocol for systems to exchange information on the locks that each transaction is waiting for), and it **does not work with SSI** (Serializable Snapshot Isolation), since require a protocol for identifying conflicts across different systems.
- For **database-internal distributed transactions** (not XA), the limitations are not so great. **Distributed version of SSI is possible**. However, for 2PC to commit a transaction, all participants must respond. Distributed transactions have a tendency of **amplifying failures**, which runs counter to building fault-tolerant systems.

To keep several systems consistent with each other, there are alternative methods that allow us to achieve the same thing without the pain of heterogeneous distributed transactions. We will return to these in Chapters 11 and 12.

Fault-Tolerant Consensus

The consensus problem: one or more nodes may propose values, and the consensus algorithm decides which value.

A consensus algorithm must satisfy the following properties:

- Uniform agreement: No two nodes decide differently.
- Integrity: No node decides twice.
- Validity: If a node decides value v, then v was proposed by some node.
- Termination: Every node that does not crash eventually decides some value.

The **termination** property formalizes the idea of **fault tolerance**. It says a consensus algorithm must make progress. Even if some nodes fail, the other nodes must still reach a decision. Termination is a **liveness property**, whereas the other three are safety properties.

Most consensus algorithms assume there are no Byzantine faults. It may break the safety properties of the protocol. It is possible to make consensus robust against Byzantine faults as long as fewer than one-third of the nodes are Byzantine-faulty.

Consensus algorithms and total order broadcast

Total order broadcast requires messages to be **delivered exactly once**, in the **same order**, to **all nodes**.

Equivalent to performing **several rounds of consensus**: in each round, nodes propose the message they want to send next, then decide on the next message to be delivered in the total order.

Total order broadcast is equivalent to repeated rounds of consensus (each consensus decision corresponding to one message delivery):

- Due to agreement, all nodes decide to deliver the same messages in the same order.
- Due to integrity, messages are not duplicated.
- Due to validity, messages are not corrupted and not fabricated out of thin air.
- Due to termination, messages are not lost.

Single-leader replication and consensus

In single-leader replication, the key is how the leader is chosen.
If the leader is manually chosen and configured by humans, such a system can work well, but does not satisfy the termination property of consensus because it requires human intervention to make progress.
Some databases perform automatic leader election and failover, promoting a follower to be the new leader if the old leader fails. This brings us closer to fault-tolerant total order broadcast, and thus to solving consensus.
There is a problem of split brain, causing inconsistent state. Need consensus to elect a leader.
But if consensus algorithms are actually total order broadcast algorithms, and total order broadcast is like single-leader replication, and single-leader replication requires a leader…
In order to elect a leader, we first need a leader. In order to solve consensus, we must first solve consensus. How do we break out of this conundrum?

Epoch numbering and quorums

- All the consensus protocols use a leader, but don't guarantee that the leader is unique. They make a **weaker guarantee**: the protocols define an **epoch number** / **ballot number** / **view number / term number**, and guarantee that **within each epoch, the leader is unique**.
- When the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus **epoch numbers are totally ordered and monotonically increasing**.
- If there is a conflict between two different leaders in two different epochs (perhaps because the previous leader actually wasn't dead after all), then the leader with the higher epoch number prevails.
- Before a leader is allowed to decide anything, it must first check that there isn't another leader with a higher epoch number to take a conflicting decision. Truth Is Defined by the Majority. A node cannot trust its own judgment just because a node thinks that it is the leader, that does not mean the other nodes accept it as their leader.
- It must collect votes from a quorum of nodes. For every decision that a leader wants to make, it must send the proposed value to the other nodes and wait for a quorum of nodes to respond. The quorum typically(not always) consists of a majority of nodes.
- Two rounds of voting: once to choose a leader, a second time to vote on a leader's proposal.
  The **key insight** is that the **quorums for two votes must overlap**: if a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the most recent leader election.
- If the vote on a proposal has no higher-numbered epoch, the current leader can be sure it holds the leadership. It can then safely decide the proposed value.

Difference between Fault-tolerant consensus algorithms and two-phase commit.

- 2PC: The coordinator is not elected; Requires a "yes" vote from every participant.
- Fault-tolerant consensus algorithms: Only require votes from a majority of nodes.

Define a recovery process by which nodes can get into a consistent state after a new leader is elected, ensuring safety properties are met.

Limitations of consensus

Consensus algorithms are a huge breakthrough for distributed systems:

- They bring concrete safety properties (agreement, integrity, and validity) to systems.
- Remain fault-tolerant (able to make progress as long as a majority of nodes are working and reachable).
- They provide total order broadcast.
- They can implement linearizable atomic operations in a fault-tolerant way.

Limitations:

- The process of nodes voting before deciding is a kind of **synchronous replication**. Databases are often configured to use asynchronous replication for better **performance**. (Even tho some committed data potentially be lost on failover)
- Consensus systems **require a strict majority to operate**.
  A minimum of three nodes to tolerate one failure, or five nodes to tolerate two failures.
  If a network failure cuts off some nodes from the rest, only the majority portion of the network can make progress, and the rest is blocked.
- Most consensus algorithms assume a **fixed set of nodes participate in voting**, you can't just add or remove nodes in the cluster. **Dynamic membership extensions to consensus algorithms** allow the set of nodes in the cluster to change over time, but they are less understood than static membership algorithms.
- Consensus systems rely on **timeouts to detect failed nodes**. In environments with highly variable network delays, especially geographically distributed systems, transient network issues make a node falsely believe the leader failed. Although this error does not harm the safety properties, frequent leader elections harm performance.
- Consensus algorithms are particularly sensitive to **unreliable network problems**.

Membership and Coordination Services

- ZooKeeper or etcd: **distributed key-value stores** or **coordination and configuration services**. The API is similar to a database: you can read and write the value for a given key, iterate over keys. They implement a consensus algorithm.
- Application developers rarely need to use ZooKeeper directly, because it is not well suited as a general-purpose database. But HBase, Hadoop YARN, and Kafka all rely on ZooKeeper running in the background.
- ZooKeeper and etcd are designed to hold **small amounts of data** that **fit** entirely **in memory** (although they still write to disk for durability)
  That small amount of data is **replicated** across all the nodes using a **fault-tolerant total order broadcast algorithm**.

- **Total order broadcast is for database replication:** if each message represents a write to the database, applying the same writes in the same order keeps replicas consistent with each other.

ZooKeeper is modeled after **Google's Chubby lock service**, not only total order broadcast (consensus), but also an interesting set of **other features** that are useful in building distributed systems:

1. **Linearizable atomic operations:**
   Use **atomic compare-and-set** operation to implement a **lock**: if several nodes concurrently perform the same operation, only one will succeed.
   The **consensus protocol** guarantees that the operation will be **atomic and linearizable**, even if a node fails or the network is interrupted at any point. A **distributed lock** is usually implemented as a **lease** who has an **expiry time** in case the client fails.
2. **Total ordering of operations:**
   When some resource is protected by a **lock or lease**, you need a **fencing token** to prevent clients from conflicting with each other in the case of a process pause.
   The fencing token is a number that monotonically increases every time the lock is acquired. ZooKeeper provides this by totally ordering all operations and giving each operation a monotonically increasing **transaction ID (zxid)** and **version number (cversion).**
3. **Failure detection:**
   Clients maintain a **long-lived session** on ZooKeeper servers, the client and server periodically exchange **heartbeats** to check that the other node is alive.
   If the connection is interrupted or a ZooKeeper node fails, the session remains active.
   If the heartbeats cease longer than the session timeout, ZooKeeper declares the session to be dead.
   Any **locks held by a session** can be configured to be **released** when the session times out (**ephemeral** nodes).
4. **Change notifications:**
   One client reads locks and values created by another client, it can also watch them for changes.
   A client can find out when another client joins the cluster (based on the value it writes to ZooKeeper), or if another client fails (because its session times out and its ephemeral nodes disappear).
   By subscribing to notifications, a client avoids to frequently poll to find changes.

Only the linearizable atomic operations really require **consensus**. The combination of these features makes systems like ZooKeeper so useful for distributed coordination.

Allocating work to nodes

- Useful in **single-leader databases**, **job schedulers,** and similar **stateful systems**: **ZooKeeper/Chubby model** works well if you have several instances of a process or service, and one of them needs to be chosen as leader. If the leader fails, one of the other nodes should take over.

- **With partitioned resources** (database, message streams, file storage, distributed actor system, etc.), we need to decide **which partition to assign to which node**.
  New nodes join the cluster, some of the partitions are moved from existing nodes to the new nodes to **rebalance the load**. As nodes are removed or fail, other nodes take over their work.

- These kinds of tasks can be achieved using **atomic operations, ephemeral nodes, and notifications in ZooKeeper**. If done correctly, this approach allows the application to automatically recover from faults without human intervention.
  An application may eventually grow to thousands of nodes. Trying to perform majority votes over thousands of nodes is inefficient. Instead:
  ZooKeeper runs on a fixed number of nodes (3 or 5) and performs its majority votes among those nodes while supporting a potentially large number of clients.
  Thus, ZooKeeper provides **coordination (consensus, operation ordering, and failure detection)** to an **external service**.
  Data managed by ZooKeeper is slow-changing: it represents information like “the node running on 10.1.1.23 is the leader for partition 7,” it may change on a timescale of minutes or hours.
  It is not intended to store the runtime state of the application; if the application state needs to be replicated from one node to another, other tools can be used.

Service discovery

ZooKeeper / etcd are often used for **service discovery.**
Find out which IP address to connect to in order to reach a particular service.

In **cloud datacenter** environments, virtual machines come and go, you don’t know the IP addresses of services. You can configure services, when they start up they **register network endpoints** in a **service registry**, then can be found by other services.

Service discovery does not require consensus:

- DNS is the traditional way to look up the IP address for a service name, it uses multiple layers of caching to achieve good performance and availability.
  Reads from DNS are not linearizable, it is usually OK if DNS query results are a bit stale.
  It is more important that DNS is reliably available and robust to network interruptions.

Leader election does require consensus:

- If the consensus system already knows who the leader is, it can help other services discover who the leader is.
  Thus, some consensus systems support read-only caching replicas.
  These replicas asynchronously receive the log of decisions of the consensus algorithm, but do not actively participate in voting. They are therefore able to serve read requests that do not need to be linearizable.

Membership services

- **ZooKeeper as membership services** is important for building highly reliable systems, e.g., for **air traffic control.**
- **A membership service determines which nodes are active and live members of a cluster.**
- Due to unbounded network delays it's not possible to reliably detect whether another node has failed. But if you couple **failure detection** with **consensus**, nodes can come to an agreement about which nodes should be considered alive or not.
- It happens that a node is incorrectly declared dead by consensus, even though it is actually alive. But it is useful for a system to have agreement on which nodes constitute the current membership.