Beyond the textbook definition of 'a mechanism for agreement'

CONSENSUS UNDERGROUND

You can't tell a stopped node from a slow one.

- ✓ From FLP impossibility to the Paxos/Raft lineage
- ✓ PBFT, Tendermint, HotStuff: defending against nodes that lie
- ✓ PoW, PoS, and consensus when identities can't be counted

By asopitech

Contents

1 Consensus Underground: The Failure Models Behind Raft, PBFT, and Proof-of-Stake

Consensus algorithms are usually introduced as ”a mechanism by which multiple nodes agree on the same value.” That's not wrong, but it undersells what you actually run into once you implement and operate one. This four-part series works through consensus by the failure model each approach targets and how it's actually built, rather than by surface-level voting mechanics — from crash faults, through Byzantine faults, to permissionless environments where you can't even count identities. It traces one line from Paxos and Raft through DAG-BFT, and another from PoW through PoS.

1.1 You Can't Tell a Stopped Node from a Slow One

Nodes stop without warning, and when one goes quiet you cannot tell whether it crashed or is merely slow. Messages get delayed, duplicated, and reordered. A disk write can appear to succeed and then turn out to have been lost. A stale leader can keep believing it's the leader and keep acting on both sides of a network partition. In blockchains, some nodes actively lie.

So the real problem consensus solves isn't collective decision-making. It's **deciding which history counts as the official one, under imperfect communication and partial failure.**

This series works through Paxos, Raft, Viewstamped Replication, Zab, PBFT, Tendermint, HotStuff, Narwhal, Bullshark, Mysticeti, PoW, PoS, and the Avalanche family. This first part covers the groundwork that no algorithm gets to skip.

1.1.1 Consensus Isn't a ”Group Discussion”

The word ”consensus” actually bundles together three distinct problems.

1. **Deciding a leader.** Which node currently owns writes.
2. **Deciding an operation order.** Making every node agree on whether `SET x=1` or `SET x=2` happened first.
3. **Committing a decision.** Guaranteeing that a committed operation is never reverted by a later re-election or failure recovery.

The important part: leader election alone does not give you consensus.

You can elect a leader with Redis Sentinel, a Kubernetes Lease, or a ZooKeeper ephemeral node. But if that leader's writes haven't been persisted to a majority of nodes, they're lost on failover. Someone being elected and that someone's work surviving are two separate guarantees.

Conversely, approaches like EPaxos and DAG-BFT skip a fixed leader entirely and derive ordering from dependencies between operations or from certificates. A leader is a means, not the goal.

1.1.2 No Response Doesn't Mean Dead

Node A sends a request to node B and gets no response. There are several possible causes:

- B's process has crashed
- B's OS has stopped responding
- the path to B is down
- B is stalled on GC or I/O
- communication only broke in the A→B direction
- the response was sent but lost in transit

In an asynchronous network, you cannot, in finite time, fully distinguish "stopped" from "delayed." That is the practical meaning of the FLP impossibility result: in a fully asynchronous system where even one node might crash, no deterministic consensus algorithm can guarantee both safety and termination.

Real systems work around this constraint by assuming one of:

- some degree of synchrony, enforced via timeouts
- randomization
- a failure detector
- pausing progress temporarily while preserving safety

Raft and HotStuff assume, as a practical matter, a partially synchronous model where the network eventually behaves for long enough stretches. In other words, these algorithms give up on always making progress. What they give up is liveness; what they keep is safety.

1.1.3 Split Brain, and the Side Effects That Leak Outside

If the network splits into two and both halves believe they are the legitimate cluster, history forks.

```
            partition
   A   B    |   C   D   E
 leader     |   new leader
```

If only the majority side is allowed to write, the left side has 2 votes and the right has 3, so only the right side can keep operating. That's the basis of majority quorums.

But this only protects the inside of the cluster. The old leader can keep issuing commands to storage, job queues, external APIs, or physical equipment. A vote taken inside the cluster never reaches those external resources.

That's why production systems don't rely on a plain "I am the leader" flag. They attach a monotonically increasing generation number — a term, epoch, or generation, i.e. a **fencing token** — and make the side-effect target itself check that token.

```
class FencedResource:
    def __init__(self):
        self.highest_token = 0
        self.value = None

    def write(self, token: int, value: str) -> None:
        if token < self.highest_token:
            raise RuntimeError("stale leader")
        self.highest_token = token
        self.value = value
```

Keeping the consensus result inside the cluster isn't enough — the generation number has to propagate all the way to wherever the side effect actually executes.

1.1.4 A Committed Log Is Not the Same as Applied State

A typical replicated state machine breaks into these layers:

```
client request
    ↓
 proposal
    ↓
 replicated log
    ↓
 commit
    ↓
 state-machine apply
    ↓
client response
```

Being appended to the log, being replicated to a majority, and being applied to the state machine are three separate events.

```
if replicated_to_majority(entry):
    commit_index = entry.index

while applied_index < commit_index:
    applied_index += 1
    state_machine.apply(log[applied_index])
```

Blur this boundary and you get bugs like:

- returning success to the client for a value that isn't committed yet
- double-applying the same command after a restart
- a mismatch between the snapshot boundary and the log
- a client retry re-executing a request that was already applied but crashed before responding

That last point matters: consensus does not automatically give you exactly-once semantics. What it gives you is log ordering and commitment. Making sure the same request doesn't execute twice is a separate concern, handled by deduplicating on a client request ID.

```
def apply(command):
    if command.request_id in completed:
        return completed[command.request_id]

    result = execute(command)
    completed[command.request_id] = result
    return result
```

`completed` is itself part of the state machine and has to be included in snapshots. Omit it, and you get a hard-to-reproduce bug that only shows up as double-application after a restart.

1.1.5 Takeaways

- What consensus actually solves isn't agreement for its own sake — it's deciding which history counts as canonical under imperfect communication and partial failure.
- Leader election, ordering, and commitment are three separate problems. Electing a leader doesn't protect you if its work isn't persisted to a majority before failover.
- Asynchronous systems can't distinguish a stopped node from a slow one (FLP impossibility). Practical algorithms assume partial synchrony or similar, trading liveness for safety.
- Majority quorums only protect the inside of the cluster. External resources need fencing tokens to reject stale generations.
- Appending to the log, committing, and applying are distinct events. Exactly-once semantics live outside consensus, as request-ID deduplication.

1.2 The Crash-Fault Lineage: Paxos and Raft

Last time we established that you can't tell a stopped node from a slow one, that majority quorums only protect the inside of a cluster, and that a committed log and applied state are different things. This time we build on that foundation and look at the first lineage of algorithms — the ones built for the crash-fault model, where nodes stop but never lie.

1.2.1 Paxos: The Only Trick Is Quorum Intersection

Paxos's job is to safely decide on a single value among multiple nodes that might crash.

Every proposal carries a monotonically increasing number. Once a majority has accepted a higher-numbered proposal, they reject anything older. The basic structure is two phases: Prepare/Promise and Accept/Accepted.

```
class Acceptor:
    def __init__(self):
        self.promised = -1
        self.accepted_number = None
        self.accepted_value = None

    def prepare(self, number):
        if number <= self.promised:
            return None
        self.promised = number
        return self.accepted_number, self.accepted_value

    def accept(self, number, value):
        if number < self.promised:
            return False
        self.promised = number
        self.accepted_number = number
        self.accepted_value = value
        return True
```

After a proposer collects Promises from a majority, if any of them already accepted a value, the proposer must adopt the highest-numbered one of those instead of its own. That's the heart of Paxos: a proposer doesn't push through its own value — it defers to whatever might already have been decided.

```
responses = [a.prepare(n) for a in acceptors]
responses = [r for r in responses if r is not None]

if len(responses) >= quorum:
    previously_accepted = [r for r in responses if r[0] is not None]
    if previously_accepted:
        value = max(previously_accepted)[1]

    accepted = sum(a.accept(n, value) for a in acceptors)
    decided = accepted >= quorum
```

The safety argument is simple. The only property Paxos relies on is that majority quorums always intersect. With 5 nodes, a majority is 3, and any two 3-node subsets share at least one node. That overlapping node is what carries a previously-accepted candidate forward into the next proposal.

1.2.1.1 Basic Paxos and Multi-Paxos

Basic Paxos decides a single value. A database needs a continuous log, so in practice you use Multi-Paxos: while

a stable leader holds, Prepare is skipped, and each log slot repeats only the Accept phase.

```
leader
   ACCEPT slot=101 value=A
   ACCEPT slot=102 value=B
   ACCEPT slot=103 value=C
```

Once you fold in that optimization, Multi-Paxos and Raft end up running almost the same way. The main difference is how leader election and log-consistency rules are specified. Comparative studies have found that Raft and Paxos share most of their structure, with leader election as the primary point of divergence.

Google's Spanner and Chubby lineage replicate data per Paxos group. But modern implementations aren't textbook Basic Paxos — they're bespoke Multi-Paxos variants layered with persistence, leader leases, reconfiguration, snapshotting, and batching.

1.2.2 Raft: Breaking the Same Safety Property into Something Implementable

Raft's goal is to restructure Paxos-equivalent crash tolerance into a form implementers can actually reason about. Raft's own site describes it as offering the same fault tolerance and performance as Paxos, designed for understandability.

Raft splits the problem cleanly into five parts: leader election, log replication, safety, membership changes, and log compaction.

```python
from dataclasses import dataclass

@dataclass
class Entry:
    term: int
    command: bytes

class RaftNode:
    def __init__(self, node_id, peers):
        self.node_id = node_id
        self.peers = peers
        self.state = "follower"
        self.current_term = 0
        self.voted_for = None
        self.log = []
        self.commit_index = -1
        self.last_applied = -1
```

The minimum state that must be durable is `currentTerm`, `votedFor`, and `log[]`. Responding before these are safely written to disk can lead to voting for two