

# Consensus Protocols



COMS4113

## COMS4113: Fundamentals of Distributed Systems

Instructor: Hubertus Franke

Columbia University

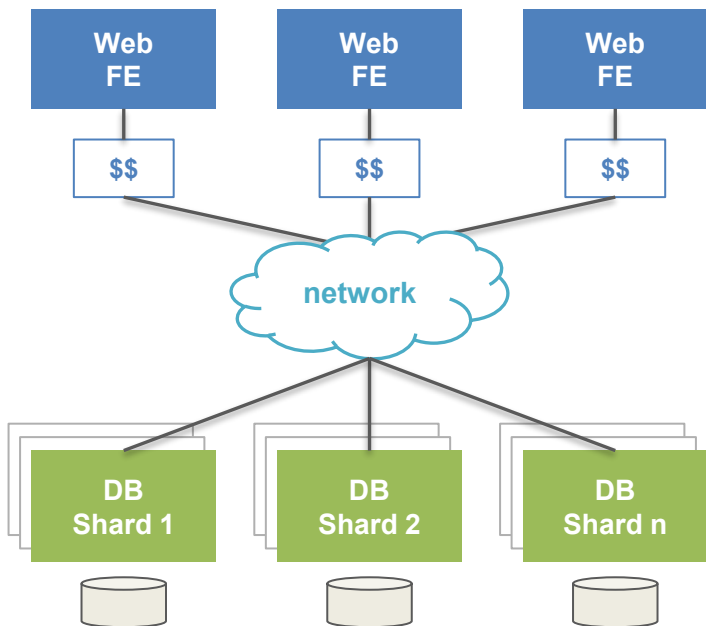
<https://frankeh.github.io/cu-ds1-2026-fall>

# Consensus Protocols

---

Agreeing on a value, and on the order of a log

The service:



Web front ends over a network; the database sharded across  $n$  servers, each shard on its own disk.

## Where we are

Lecture 8 will give us **2PC** for the *sharding* half of this picture - atomic commitment across shards. It ended on a negative result: 2PC **blocks**, because the coordinator is a single point of failure.

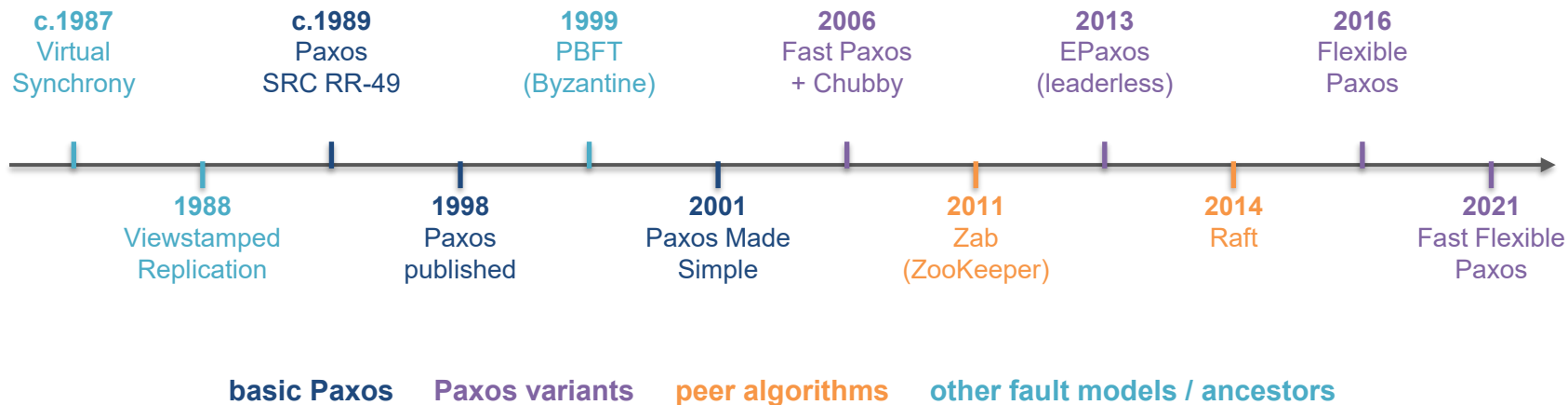
## Today: the replication half

Each shard becomes a **replica group**. Its replicas must agree on the *order* of operations - that is **consensus**, and it is what makes a fault-tolerant coordinator possible.

- The problem: **what** to replicate, with **what** semantics, and **how**
- **A first attempt** built from 2PC (Phase Commit) - and why it fails
- **Paxos**: the protocol, its three phases, and seven worked scenarios (*the bulk of today*)
- **Multi-Paxos**: from a write-once register to a replicated log
- The **Paxos family**: Fast, Flexible, Fast Flexible, Egalitarian
- **Other families**: Viewstamped Replication, Zab, and the Byzantine case
- **Raft**, and how it compares to Paxos in theory and in production

# Consensus is a family of protocols

"Paxos" names a **family**, not one algorithm - and it is not the only family. Here is the shape of it.



## You are here

The next ~25 slides are **basic Paxos** - the one you will implement. **Raft is not a Paxos variant**: it is a separate algorithm with the same guarantees. Both come at the end.

# Some forward looking statements (1)

- In distributed, transactional storage systems, **2PL**, **WAL**, and **2PC** are foundational protocols used to guarantee the ACID properties (Atomicity, Consistency, Isolation, Durability) across single nodes and distributed shards.
- **2PL (Two-Phase Locking)**
  - A concurrency control protocol used to ensure serializability (isolation) for transactions operating on a single shard.
  - How it works:
    - **Growing Phase:** The transaction acquires locks (shared locks for reads, exclusive locks for writes) as needed. No locks can be released during this phase.
    - **Shrinking Phase:** Once the first lock is released, the transaction enters the shrinking phase where it can only release locks and cannot acquire any new ones.
  - Impact:

By preventing transactions from interleaving conflicting operations, 2PL ensures that concurrent execution yields the same result as if the transactions were run one after another.

# Some forward looking statements (1)

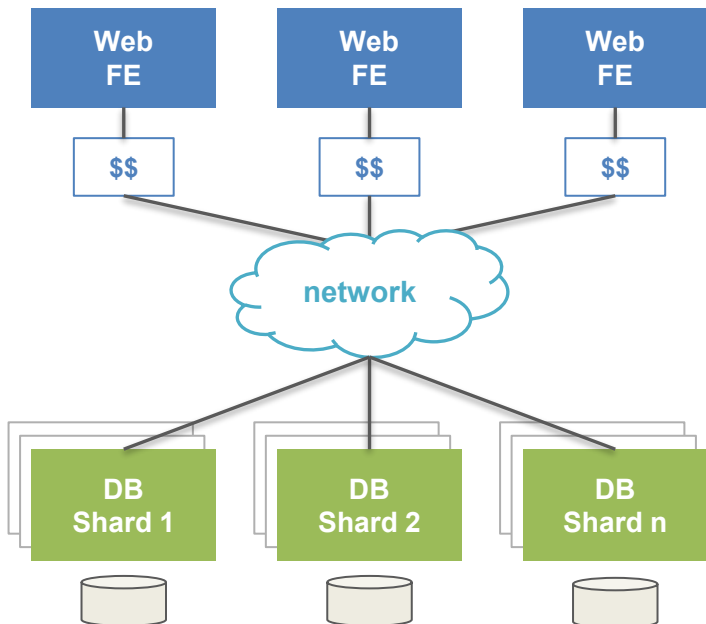
- **WAL (Write-Ahead Logging)**
  - A recovery and durability mechanism where modifications are recorded in an append-only log on stable storage *before* the actual data pages are updated on disk.
  - Whenever a transaction modifies data, the change description is written to the WAL first. If the system crashes, the storage engine can read the WAL during restart to **REDO** committed transactions (ensuring durability) or **UNDO** uncommitted transactions (ensuring atomicity).
  - It decouples fast sequential disk/SSD writes from random I/O database page updates, providing high performance alongside crash recovery guarantees.
- **2PC (Two-Phase Commit)**
  - A distributed consensus protocol used to coordinate transaction outcomes across multiple independent shards or nodes.
  - 2 Phases
    - **Prepare Phase:** The coordinator node asks all participating shards if they are ready to commit. Each shard locks its data, writes its local WAL, and votes "Yes" (prepared) or "No" (abort).
    - **Commit/Abort Phase:**
      - If *all* shards vote "Yes", the coordinator instructs everyone to **Commit**.
      - If *any* shard votes "No" (or fails), the coordinator instructs everyone to **Abort**.
  - It ensures atomic commitment across a distributed boundary—either all shards commit the transaction, or all of them roll it back, preventing partial updates.

# The Problem

---

Fault tolerance needs replication; replication needs agreement

# Example: Transactional Web Service



## What we have

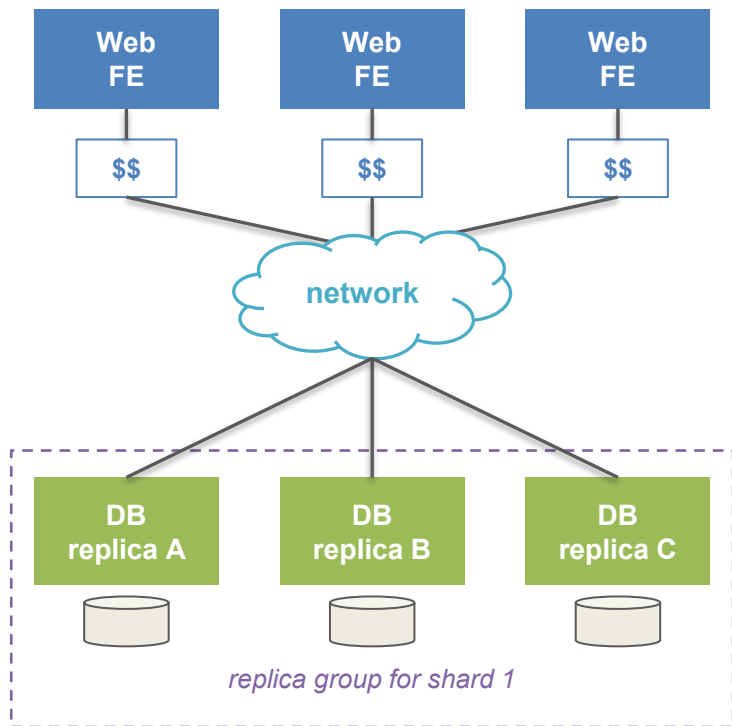
Sharded, transactional storage: each shard runs 2PL (2-phase locking) and a WAL (Write Ahead-Locking) [Lecture-7]), and 2PC spans them [Lecture-8].

## Fault tolerance problems without replication

Each shard's data and WAL sit on **one disk**. If that disk dies, the shard is gone - and with it any transaction that touched it.

If the *machine* dies, the shard is unavailable until it returns, and 2PC blocks meanwhile.

# Solution: Replication



## Each shard becomes a replica group

Keep **copies** of each shard's data and WAL on several machines - so the shard can **survive a disk, machine or rack failure**.

## Which raises three questions

1. *What* state do we replicate?
2. With *what* semantics?
3. *How* do we keep the copies in step?

# Question 1: What state to replicate?

- The disk image?
- The in-memory image?
- The **WAL**?
- The locks?
- ... anything else?

# Basic answer: replicate the WAL

- **Claim:** if all replicas execute **all WAL ops, in the same order**, then every other piece of state - the DB image, the locks, the caches - is reconstructed identically on each replica.
  - The assumption that makes the claim true: the operations are **deterministic**.
- This is **state machine replication**: same start state, same sequence of inputs, same end state.
- *(It is also useful to ship DB checkpoints to a recovering or brand-new replica. We ignore that here and focus on the log.)*

*Note what the claim reduces the problem to: not "replicate a database" but "agree on a sequence".*

## Question 2: What Semantics?

- **Requirement:** all replicas apply **(1) the same log entries, (2) in the same order.**
- Drop either half and the replicas diverge:
  - One replica **skips** an entry the others apply - permanently missing an update.
  - Two replicas apply two updates to a row **in opposite orders.**
- Hence, the semantics we need are: **agreement on a sequence.**

# Question 3: How to replicate?



COMS4113

- **Requirement (restated):** all replicas apply the same log entries, in the same order.
- **One idea: use 2PC.** We already have it, it already makes several machines agree on a decision, and the class already knows it.
- So let us build the obvious thing out of 2PC - and then break it.
- *What breaks is exactly what consensus fixes.*

# A First Attempt

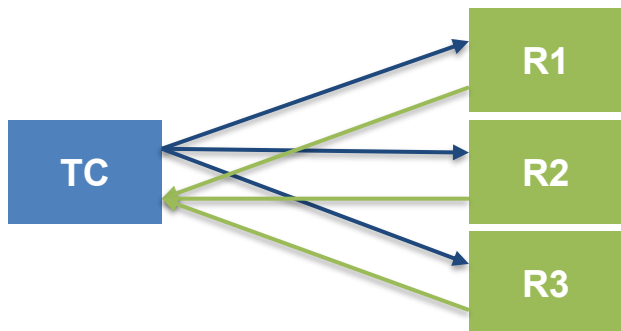
---

Build replication out of 2PC, then break it

# A First Attempt: 2PC per log entry

## Prepare Phase

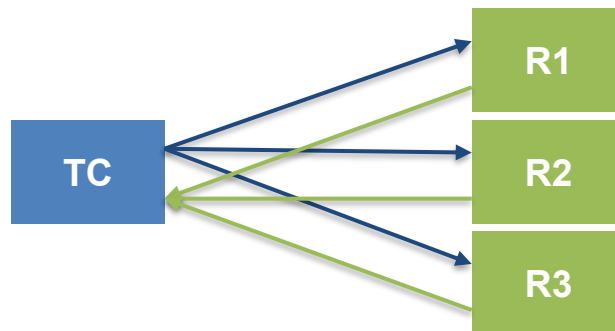
1. PREPARE



2. PREPARE-OK / FAIL

## Commit Phase

3. COMMIT / ABORT



4. OK

### The idea

One replica is the **TC**. For **each log entry** it runs a full 2PC with the others as participants, finishing one round before the next.

*TC = transaction coordinator ; R1/R2/R3 = three replicas of one shard.*

# The First Attempt, Precisely

- One replica is assigned as **TC**. The TC decides the order of ops in the log, and runs **2PC for each log entry** - blocking each time until that round finishes before launching the next.
- This does give us both halves of what Question 2 demanded:
  - All replicas apply **all** log entries - thanks to 2PC's atomicity.
  - All replicas apply them **in the same order** - thanks to the TC pushing entries strictly sequentially.
- It can be optimized to run 2PC over **batches** of log entries. *When would you need to push a batch?*

*So far this looks like it works - which is why it is worth taking seriously before we break it.*

# Why the First Attempt Fails

## Problem 1: it is NOT fault tolerant

The TC must wait for **every** replica to reply before it can proceed. So it blocks whenever **one** replica is slow, disconnected, or dead.

We added replicas *to survive failures*, and built something that stops when any one of them fails.

## It is more durable than 2PC-across-shards, though

Credit where due: the data now exists in several places, so a lost disk no longer loses the shard.

## Problem 2: who becomes TC when the TC dies?

Someone must. But there must be **at most one** - two TCs would impose different orders on the log, and the replicas diverge. That is **leader election**, and 2PC does not have it: 2PC *assumes a static coordinator*.

# Consensus Protocols

---

Majorities instead of unanimity

# What Consensus Gives You

- A **consensus protocol** lets a set of nodes agree on a value, and keeps agreeing even while some of them are down.
- The move that makes this possible: require a **majority**, not unanimity.
  - 2PC needed *every* participant to vote yes - so one failure stopped it. That was Problem 1.
  - A majority protocol tolerates  **$f$  failures with  $2f+1$  nodes**: 1 of 3, 2 of 5.
- Because any two majorities **overlap**, no two of them can ever decide different values. That overlap is the whole safety argument.

*Everything else in this lecture is machinery on top of that one idea.*

- Any two majorities of the same node set **intersect** in at least one node.
- With  $N = 2f+1$  nodes, a majority is  $f+1$ , and **the protocol makes progress while any  $f$  are down**.
- The intersecting node is what carries information forward: a new proposer asking a majority is **guaranteed** to hear about any value a previous majority already accepted.
- This is why we can replace the static coordinator. A new coordinator does not need the old one - it needs a **majority**.

# Two Protocols We Will Study



COMS4113

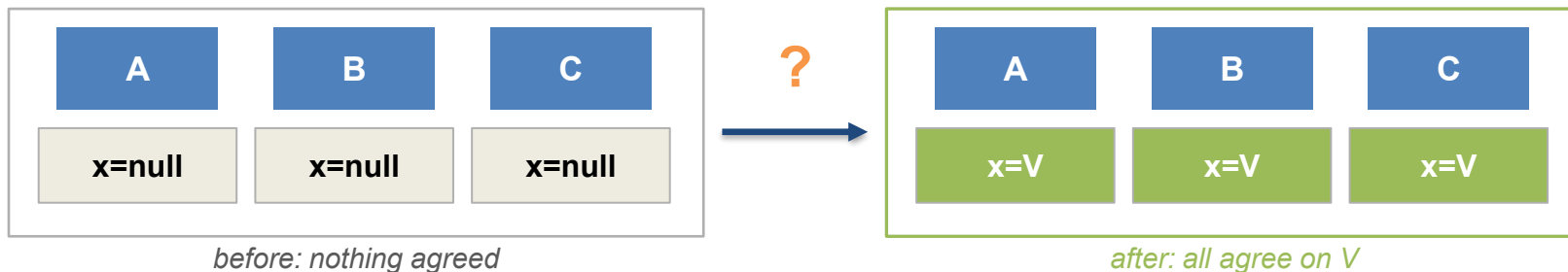
- **Paxos** [Lamport-1998] - the original. Solves the *basic* consensus problem from the Agreement lecture: consensus on the value of a **write-once register**, with consistency, validity and termination requirements.
- **Raft** [Ongaro-2014] - more recent, operates at a higher level of abstraction, and shows very directly how to replicate a WAL to get fault-tolerant transactions.
- **Raft is not a Paxos variant.** It is a *separate* algorithm - "equivalent to Paxos in fault-tolerance and performance", but decomposed differently. Same problem, independent design.
- We study **Paxos** in depth and extend it to the WAL, then come back to Raft, to the Paxos variants, and to the other families at the end.

# Paxos

---

Consensus on a single write-once register

Paxos solves the generic problem of consensus: **N nodes want to agree on the value of a *write-once register*.**



## The protocol guarantees

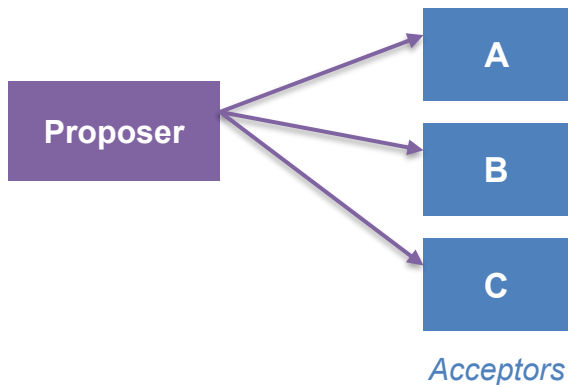
1. **consistency** - all non-faulty nodes choose the same value.
2. **validity** - the chosen value was actually proposed by some proposer.

## The protocol is likely to achieve but does NOT guarantee

3. **termination** - eventually, a value is chosen.

# Overview: the two roles

Every node can fulfil **two roles** - and in general plays both at once.



## Proposers

Issue a series of **rounds** of proposals, ordered by a logical clock (the proposal number  $N$ ).

## Acceptors

Accept or reject a proposer's values according to the rules of the protocol - and **remember** what they accepted.

## Why any node can be a proposer

There is no elected leader in basic Paxos. If a proposer stops, **another node just starts a new round** - which is exactly the leader-election problem the first attempt could not solve.

# The Protocol: State at Each Node

- **N<sub>p</sub>** - the highest proposal number **seen** to date (*initially nil*)
- **N<sub>a</sub>** - the highest **accepted** proposal number (*initially nil*)
- **V<sub>a</sub>** - the **value** of that highest accepted proposal (*initially nil*)
- **Done** - whether consensus has been reached (*initially false*)
- The protocol has three phases: **Propose**, **Accept**, **Decide**.

*Four variables per node. Every rule in the next three slides reads or writes exactly these.*

# Phase 1: Propose

- **@Proposer** (*assumes Done = false*)
  - Choose a new proposal number  **$N > N_p$** .
  - Send  $\langle \text{PROPOSE}, N \rangle$  to the acceptors - **including itself**.
  - Wait until a **majority** of acceptors return PROPOSE-OK. On timeout, back off and restart Paxos.
- **@Acceptor** - on receiving  $\langle \text{PROPOSE}, N \rangle$ :
  - **If  $N > N_p$  then:** set  $N_p = N$ , and reply  $\langle \text{PROPOSE-OK}, N_a, V_a \rangle$ .
  - (*Otherwise ignore, or reject with the higher  $N_p$  it has seen.*)

*Phase 1 is how a proposer both claims a round number **and** learns what has already been accepted.*

# Phase 2: Accept

- **@Proposer** (*assumes PROPOSE-OKs from a majority*)
  - Choose **V** := the value of the **highest-numbered proposal** among those returned by the acceptors - **or any value** if no  $V_a$  came back.
  - Send  $\langle \text{ACCEPT}, N, V \rangle$  to all acceptors, including itself.
  - Wait until a majority return ACCEPT-OK. On timeout, back off and restart.
- **@Acceptor** - on receiving  $\langle \text{ACCEPT}, N, V \rangle$ :
  - **If  $N \geq N_p$  then:** set  $N_p = N$ ,  $N_a = N$ ,  $V_a = V$ , and reply  $\langle \text{ACCEPT-OK} \rangle$ .

*The highest- $N_a$  rule is what makes a new proposer continue the old one's work instead of overriding it.*

# Phase 3: Decide



COMS4113

- **@Proposer** (*assumes ACCEPT-OKs from a majority*)
  - Send **Done** to the client - consensus has been reached.
  - Send  $\langle \text{DECIDE}, N, V \rangle$  to all acceptors, including itself. (*It can keep resending until all reply - but acceptors can also learn the decision from other proposers.*)
- **@Acceptor** - on receiving  $\langle \text{DECIDE}, N, V \rangle$ :
  - **If  $N \geq N_p$  then:** set  $N_p = N$ ,  $N_a = N$ ,  $V_a = V$ , reply  $\langle \text{DECIDE-OK} \rangle$ , set **Done = true** and terminate Paxos.

*Consensus is reached at the end of phase 2. Phase 3 only tells people - which is why it is off the critical path.*

# The whole protocol on one page

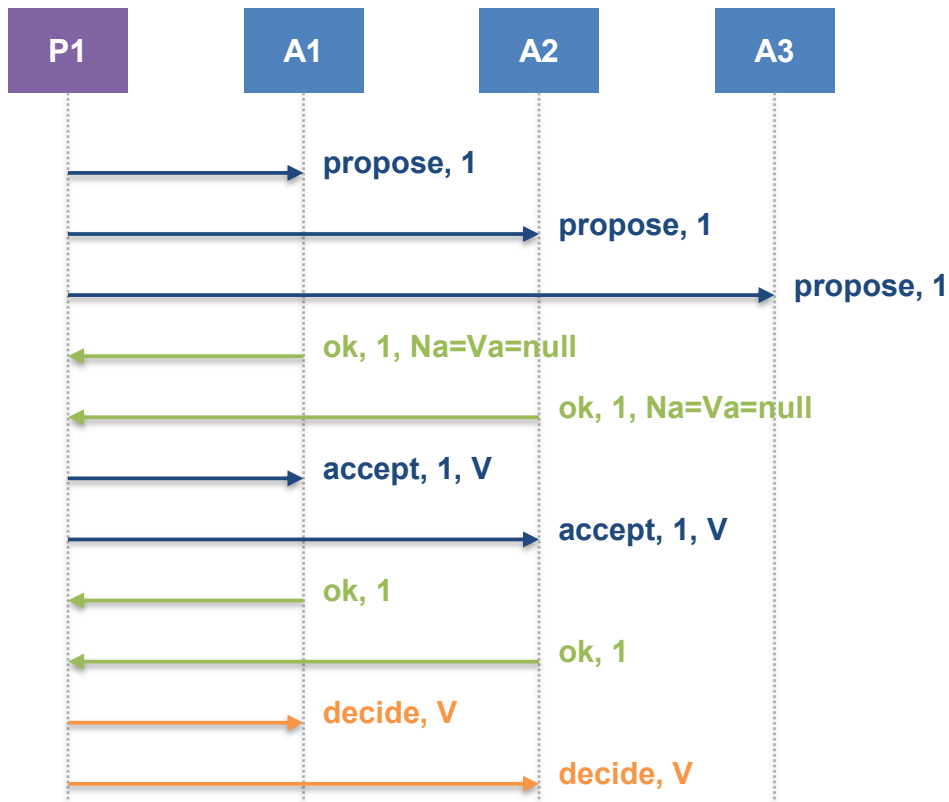
|                                                      | Proposer                                          | Acceptor                                          | Guard        |
|------------------------------------------------------|---------------------------------------------------|---------------------------------------------------|--------------|
| 1. PROPOSE                                           | pick $N > N_p$ , send to all                      | reply <b>PROPOSE-OK</b> , $N_a$ , $V_a$           | $N > N_p$    |
| 2. ACCEPT                                            | $V := V_a$ of highest $N_a$<br>(else free choice) | set $N_p, N_a = N$ , $V_a = V$ ; <b>ACCEPT-OK</b> | $N \geq N_p$ |
| <i>a majority of ACCEPT-OK = the value is CHOSEN</i> |                                                   |                                                   |              |
| 3. DECIDE                                            | tell client and all acceptors                     | set Done; terminate                               | $N \geq N_p$ |

## The two rules to memorize

**Phase 1 asks, phase 2 tells.** And a proposer may only propose the value it *learned* - free choice needs an empty majority.

- Paxos is best understood by reading the protocol, then an example, then **the protocol again**, then another example, ...
- We work through **seven scenarios**: one proposer, concurrent proposers, sequential proposers, and four failure cases.
- In every one of them, ask yourself the same two questions:
  - *Which value gets chosen - and could a different majority have chosen a different one?*
  - *Which rule of the protocol is doing the work here?*
- Re-read the protocol at home and build your own scenarios, trying to break it. That is the best way to understand it - and you will need it when you implement it.

# 1. Single Proposer



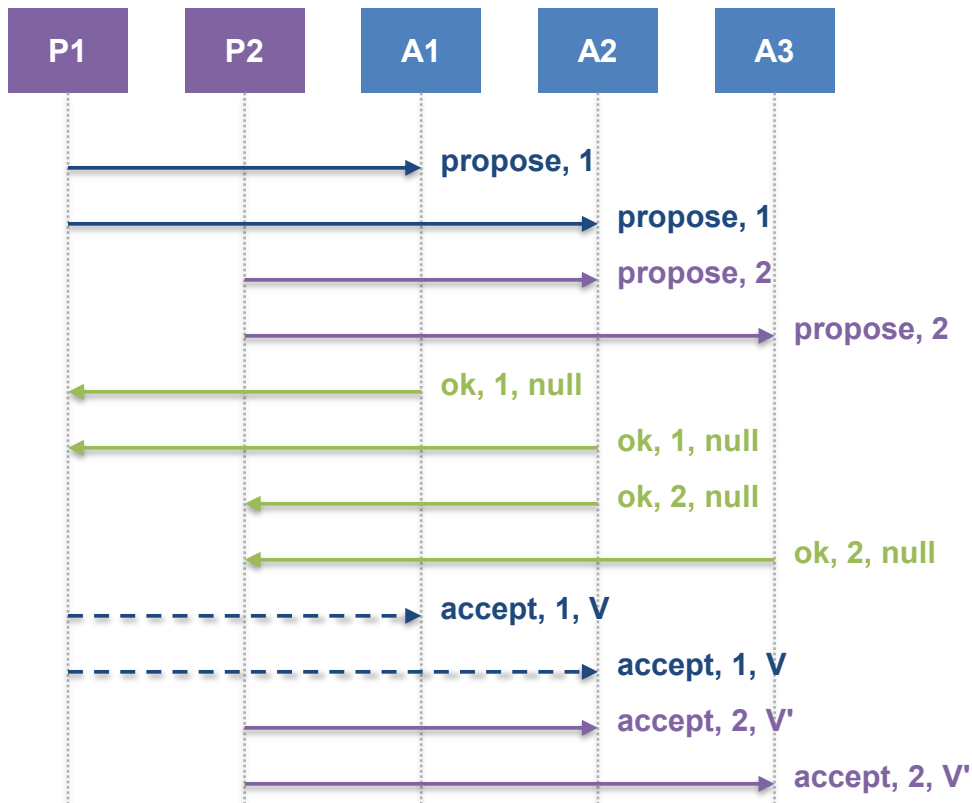
## What happens

The uncontended case. A **majority of 2 of 3** suffices at every step - A3 need never reply, and the value is still chosen.

## Outcome

Consensus on **V**. Note A3 was never required.

## 2. Concurrent Proposers



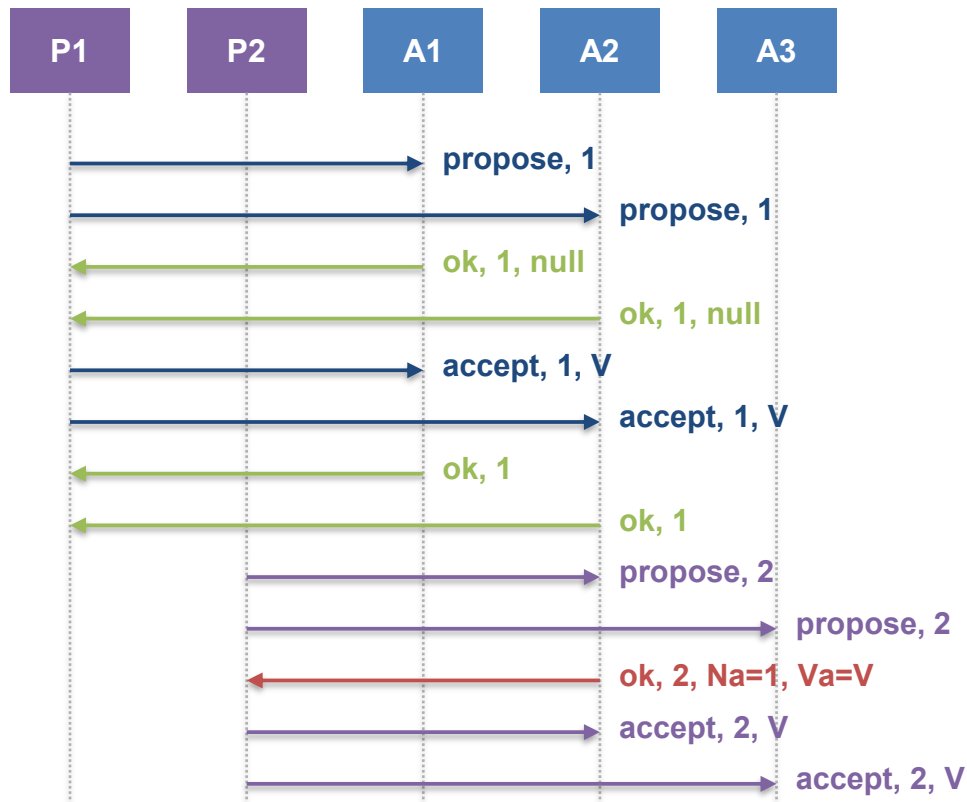
### What happens

Both get a majority in phase 1. But A2 promised  $N_p=2$ , so it **rejects** P1's accept (dashed) and P1's round dies.

### Outcome

Consensus on **V'**; P1 must retry with a higher N.

# 3. Sequential Proposers



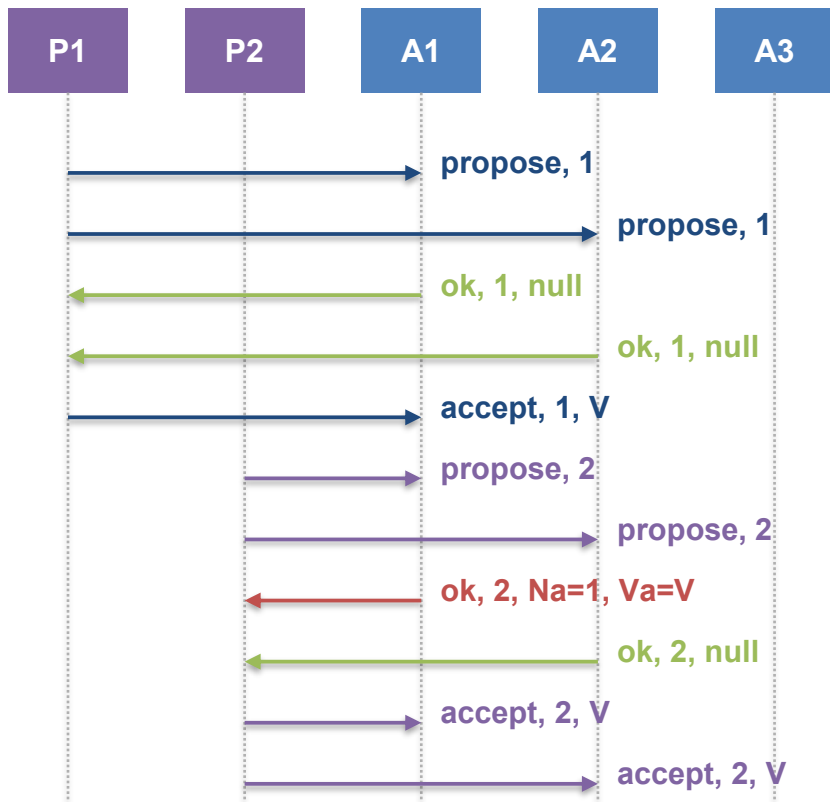
## What happens

P1 finishes first. When P2 runs phase 1, **A2 reports  $N_a=1$ ,  $V_a=V$** , so the highest- $N_a$  rule forces P2 to propose **V**.

## Outcome

**V remains the consensus value.** This is the rule that makes Paxos safe.

# 4. Proposer Fails Mid-Round



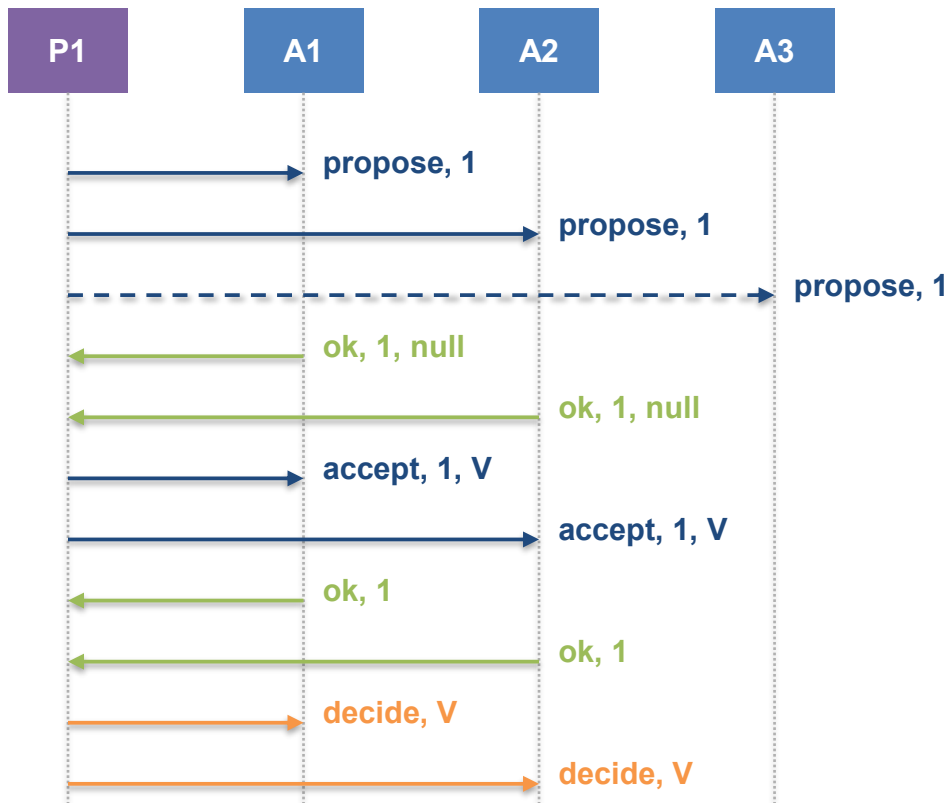
## What happens

P1 dies after **one** accept - not a majority, so nothing was chosen. P2 takes over and A1 still reports **V**.

## Outcome

P2 adopts **V** anyway. A partial round is *not* lost information.

# 5. Acceptor Fails



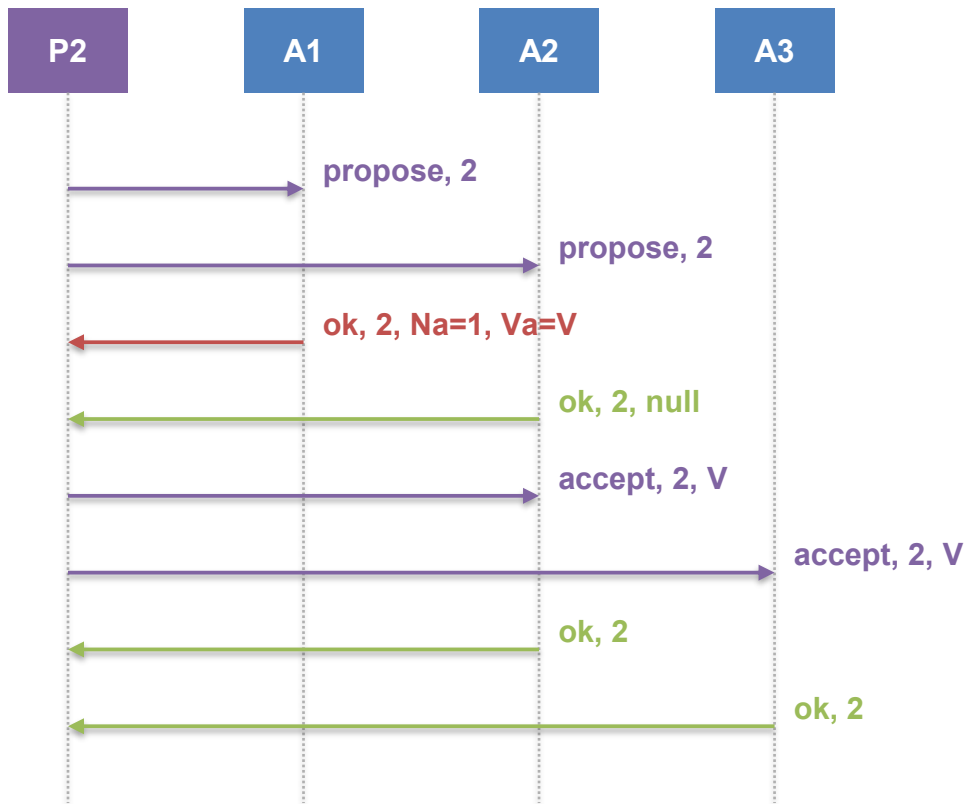
## What happens

A3 is down for the whole run (dashed = no reply). With  $N=3$  a majority is 2, so  $f = 1$  **failure is tolerated**.

## Outcome

Consensus on **V**, unaffected. This is the fault tolerance we came for.

## 6. Different Majorities per Phase



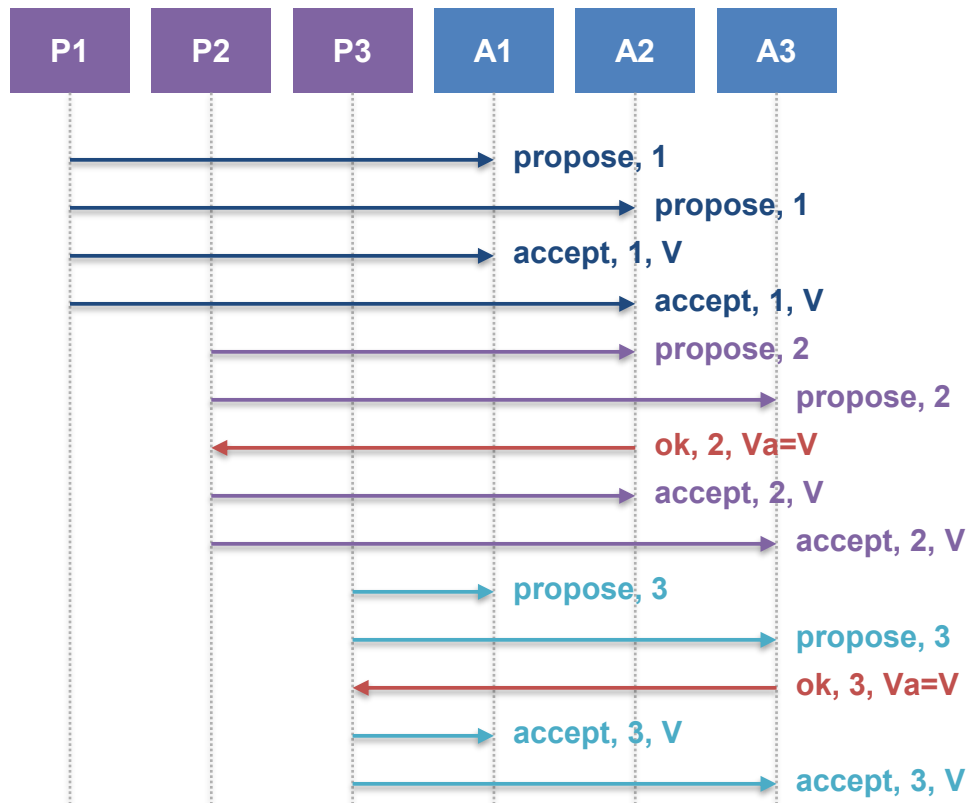
### What happens

P2's two majorities **differ**: {A1,A2} in phase 1, {A2,A3} in phase 2 - yet **V** is still chosen, because they intersect at A2.

### Outcome

Intersection, not identity, is what safety needs. (*Hold that thought - Flexible Paxos.*)

# 7. Three Proposers in Sequence



## What happens

Three proposers, one after another, each learning **V** from a different acceptor: P2 from A2, P3 from A3.

## Outcome

**V** survives all three. Once chosen, a value cannot be un-chosen.

# Paxos Is Fault-Tolerant

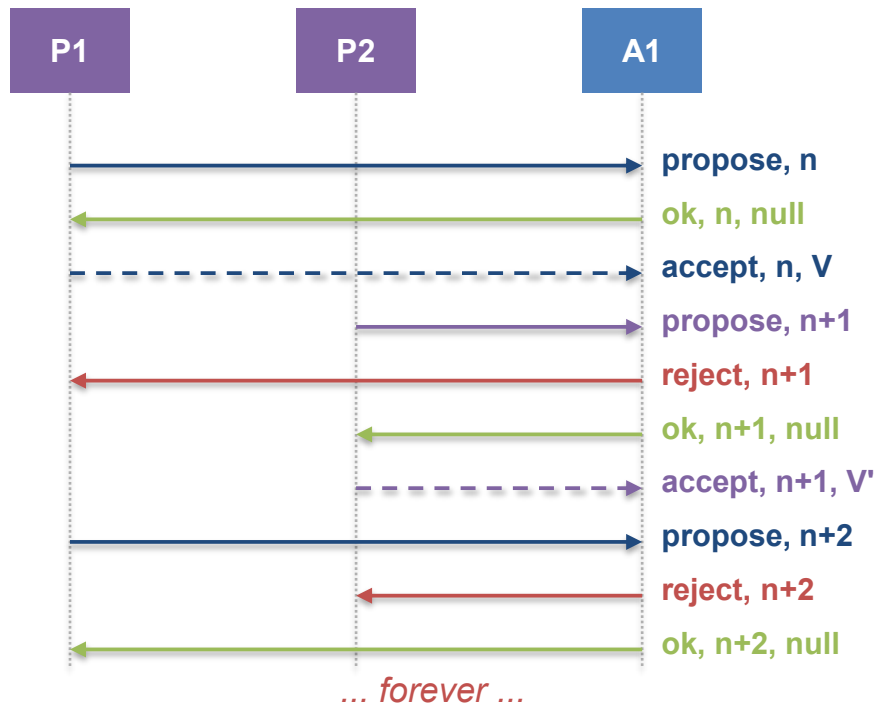


COMS4113

- A Paxos **run** (aka **instance**) consists of **one or more rounds**, conducted by possibly different proposers. New proposers *continue the work of* previous ones - that is the key to fault tolerance.
- If a proposer dies, another times out and offers to be proposer. Because of the highest-**Na** rule, the new proposer will keep propagating any **already-chosen value**.
- A run succeeds if **at least a majority** of nodes are up and accept the proposal - so **f** failures are tolerated with **2f+1** nodes.
- Compare the first attempt: there, one slow replica stopped everything. **Both** of its problems are now solved.
- But there are degenerate cases where **Paxos does not finish** - next slide. These can be made unlikely with random back-offs.

*Fault tolerance comes from the majority rule; safety across proposer changes comes from highest-**Na**.*

# Paxos May Not Terminate



## Dueling proposers

Each proposer's phase 1 invalidates the other's phase 2 - so they make progress on *round numbers* and none on choosing a value. Paxos is **safe** but not **live**.

## The fix

**Random backoff** - which Raft promotes to a first-class mechanism (see its leader-election slide).

# Extending to Read-Write: Multi-Paxos

- The protocol so far solves consensus on a **write-once register**.
- Real data structures are rarely write-once - so at first sight this looks useless for replicating a database.
- But many useful structures **reduce to a set of write-once registers**.
- **The WAL is exactly that:** a long vector of write-once registers. Each index in the log (record #1, #2, ...) is written once and never updated.
- So: run **one instance of Paxos per log index**. That is **Multi-Paxos**.

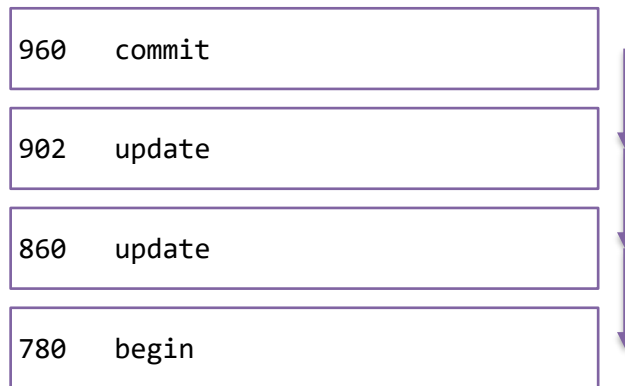
*This is the step that turns a theoretical result into a replicated database.*

# Log Records: begin / commit / abort

|         |                                                                                                                       |
|---------|-----------------------------------------------------------------------------------------------------------------------|
| LSN     | <b>Log Sequence Number</b> — usually implicit: the address of the first byte of the log entry.                        |
| prevLSN | LSN of the <b>previous record for this transaction</b> — a linked list threading each transaction's records together. |
| xID     | Which transaction this record belongs to.                                                                             |
| type    | The operation: begin, commit, abort, update.                                                                          |

The LSN is an **address**, not a counter: it identifies where the record sits in the log, so a page can cheaply record the last LSN that touched it (slide 32).

## The prevLSN chain



*Each record points back at the same transaction's previous one — this is how undo walks a transaction backwards.*

# Example: Replicating the WAL

## One Paxos instance per index:

|     |          |       |                |
|-----|----------|-------|----------------|
| 780 | prevLSN: | 0     | <i>inst. 1</i> |
|     | xId:     | 42    |                |
|     | type:    | begin |                |

|     |          |        |                |
|-----|----------|--------|----------------|
| 860 | prevLSN: | 780    | <i>inst. 2</i> |
|     | xId:     | 42     |                |
|     | type:    | update |                |

|     |          |        |                |
|-----|----------|--------|----------------|
| 902 | prevLSN: | 860    | <i>inst. 3</i> |
|     | xId:     | 42     |                |
|     | type:    | commit |                |

### Start empty

No value is assigned to any index yet.

### One instance per index

For each index, Paxos makes the replicas agree on **which record** goes there - begin, update, commit, prepare.

### Appending a record

A replica picks an **unused index** and proposes. If no acceptor returns a value there, it may use its own record.

- The description so far is **inefficient** in two ways:
  - Clients may talk to any replica, so **proposers duel** - and a lost duel is wasted work, not just wasted time.
  - Every time a replica becomes proposer it must run **phase 1** again, just to agree on N.
- **The fix: a stable leader.** Like 2PC's coordinator - except this one **can change seamlessly** when it fails.
- That single change is what most production Paxos systems actually run, and it is also the design Raft starts from.

# Leader-based Multi-Paxos

- Say proposer **P** completes phase 1. Then P may assume the role of **Leader** for a predefined period - its **lease**.
- While P is leader:
  - Clients send requests to P, so **no duels**.
  - P runs **only phase 2** per log entry - phase 1 is amortized over all of them. Steady state costs **one round trip**.
- If P fails, another replica waits out the lease, runs phase 1, and becomes leader. **Leader election, solved** - the thing 2PC could not express.

- A **lease** gives a node permission to act in a role for a period of **real time**, renewable within that window.
- Used throughout distributed systems as an **optimization** - never for safety:
  - **Logical** clocks ( $N_p$ ,  $N_a$  in Paxos) are used for safety/correctness - they raise no synchronization challenges.
  - **Physical** time, via leases, is used only for optimization.
- **The assumption: small clock skew during the lease period.** Which is why leases cannot be long - 30-second leases are typical.
- If the assumption breaks you lose *performance*, not *correctness* - two leaders may both think they hold the lease, and Paxos's round numbers still keep them safe.

# The Paxos Family

---

Four variants, and what each one trades away

# Why There Are Variants

- Basic Paxos fixes three quantities: **majority quorums**, **three message delays** to learn a value, and **one leader** at a time.
- Each of the next four protocols relaxes exactly one of those - and **pays for it somewhere else**.
  - **Fast Paxos** - fewer message delays, larger quorums.
  - **Flexible Paxos** - smaller steady-state quorums, weaker failure-recovery availability.
  - **Fast Flexible Paxos** - the Flexible insight applied to Fast Paxos's quorums.
  - **EPaxos** - no leader at all, at the cost of dependency tracking.
- Read each slide as: *what changes -> what it buys -> what it costs*.

*There is no free lunch here. A variant that only listed its win would be teaching the wrong lesson.*

## What changes

Clients send proposals **directly to the acceptors**, skipping the leader's phase-2a step. A *fast round* replaces the classic one when there is no contention.

## What it buys

Learning takes **two message delays instead of three** (counting the client's own message). Lamport shows these are *the minimum a general consensus algorithm can achieve*.

## What it costs

**Larger quorums** and a **collision-recovery path**. With classic quorums of N-F and fast quorums of N-E, safety needs  $N > 2F$  and  $N > 2E + F$ ; recovery adds one or two delays back.

## What changes

**Majorities are not necessary.** The two phases may use **non-intersecting** quorums - intersection is required only *across* phases, not within one.

## The rule, in one line

$|Q1| + |Q2| > N$ . Equivalently, the smallest legal phase-1 quorum is  $|Q1| = N - |Q2| + 1$ . Scenario 6 already showed why intersection is what safety needs.

## What it buys, and what it costs

Shrink **Q2** for cheaper steady state - with 8 acceptors and  $Q2=4$  the authors measure **42ms -> 37ms** and **198 -> 264 reqs/sec**. On even  $N$  it is a free win. The cost: **fewer failures tolerated while recovering** from leader failure.

## What changes

The Flexible insight applied to **Fast Paxos**: its quorum intersection requirements can also be safely relaxed.

## The result

Fast Paxos's requirements reduce to **one additional intersection**: between phase-1 quorums and any *pair* of fast-round phase-2 quorums. The authors' conclusion: Fast Paxos's quorums *"are larger than necessary"*.

## Why it matters

Fast Paxos's fast path was often unusable because its quorums were too big. Shrinking them makes the two-delay path **reachable in more configurations**.

# Egalitarian Paxos (EPaxos)

## What changes

**There is no leader.** Clients may submit to *any* replica, which becomes the *command leader* for that command only. Ordering is decentralized and per-command.

## How: order only what actually conflicts

Each command carries **dependencies** on the commands it *interferes* with. **Non-interfering commands commit in one round trip**; interfering ones take a second, classic phase.

## Quorums - and the cost

With  $N = 2f+1$  the optimized fast-path quorum is  **$f + \text{floor}((f+1)/2)$**  - that is **2 of 3, 3 of 5**: a simple majority, *without* the supermajority Fast Paxos needs. The cost is a **dependency graph** to track and execute in order, including cycles.

# Other Protocol Families

## Viewstamped Replication - it PREDATES Paxos

Oki & Liskov, **PODC 1988**. The same result reached from the *replication* side: views and a primary, not ballots.

## Zab - powers ZooKeeper, and is NOT Paxos

Junqueira, Reed & Serafini, **DSN 2011**. *Atomic broadcast* that also guarantees **primary order**, which Paxos does not. ZooKeeper's docs ask "Isn't this just Multi-Paxos?" - and answer "**No**".

## PBFT - a different FAULT MODEL, not a variant

Castro & Liskov, **OSDI 1999**. Tolerates nodes that *lie*, not just crash:  **$3f+1$**  replicas, not  $2f+1$ . **Lecture 14**.

**The realistic summary:** van Renesse et al. (*IEEE TDSC 12(4)*, 2015) describe Paxos, VR and Zab in **one framework** - they are dialects. **PBFT is the real outlier**.

# Raft

---

The same guarantees, designed to be understandable

# Raft: Consensus You Can Teach

- Ongaro & Ousterhout, **USENIX ATC 2014** - and the title is the thesis: *In Search of an Understandable Consensus Algorithm*.
- Raft exists because **Paxos is hard to understand and harder to implement**. Understandability was an explicit *design goal*, not a side effect.
- Pursued by **decomposition** - leader election, log replication and safety are separate mechanisms - and by deliberately **reducing the state space**.
- The authors backed the claim with a **user study** comparing Raft and Paxos comprehension. That is a rare empirical claim to make about a protocol.
- **Same guarantees as Multi-Paxos**. The difference is what it makes easy.

*The fullest reference is Ongaro's Stanford PhD thesis, Consensus: Bridging Theory and Practice (2014).*

# Raft: Strong Leader



COMS4113

## Paxos permits any proposer at any time

Which is why duelling proposers exist, and why leader-based Multi-Paxos had to be bolted on afterwards.

## Raft **REQUIRES** a leader

All client requests and all log entries flow **through the leader**. There is no leaderless mode - the strong leader is part of the specification.

## The trade, stated plainly

Reasoning and debugging get much simpler. But the leader is a **throughput bottleneck** and its failure is a **brief stall** - which is precisely what EPaxos rejected.

# Raft: Leader Election

## The problem is the one from slide 40

Duelling proposers livelock because both keep timing out and retrying **in step**.

## Raft's answer: randomized election timeouts

Each follower waits a **random** interval before standing for election, so one candidate almost always gets there first. **Terms** number the elections, exactly as proposal numbers number Paxos rounds.

## Same trick, different status

Paxos treats random backoff as an *engineering patch* for a liveness problem. Raft makes it **part of the protocol**. That is a good illustration of what 'designed for understandability' actually buys.

# Raft: the Log IS the Abstraction



COMS4113

## Paxos reduces a log to write-once registers

Multi-Paxos runs one instance per index - the reduction from slide 41. Correct, but indirect.

## Raft replicates the log directly

The **Log Matching Property**: if two logs contain an entry with the **same index and term**, the logs are **identical in every entry up through that index**.

## Why that helps

One strong invariant replaces per-index reasoning, and it maps directly onto the problem this lecture opened with: **replicate the WAL**.

# Raft: Membership Changes

## A real operational problem basic Paxos leaves open

Machines get added, removed and replaced. Change the node set naively and **two disjoint majorities** can exist at once - old and new - which breaks safety.

## Raft's answer: joint consensus

Move through a **transitional configuration** requiring majorities in *both* the old and the new node sets, so no two disjoint majorities can ever decide independently.

## And a note on which Raft you read

Ongaro's **thesis** replaces joint consensus with a simpler **single-server change** - add or remove one node at a time. If you implement from the thesis, expect that difference.

# Paxos vs Raft



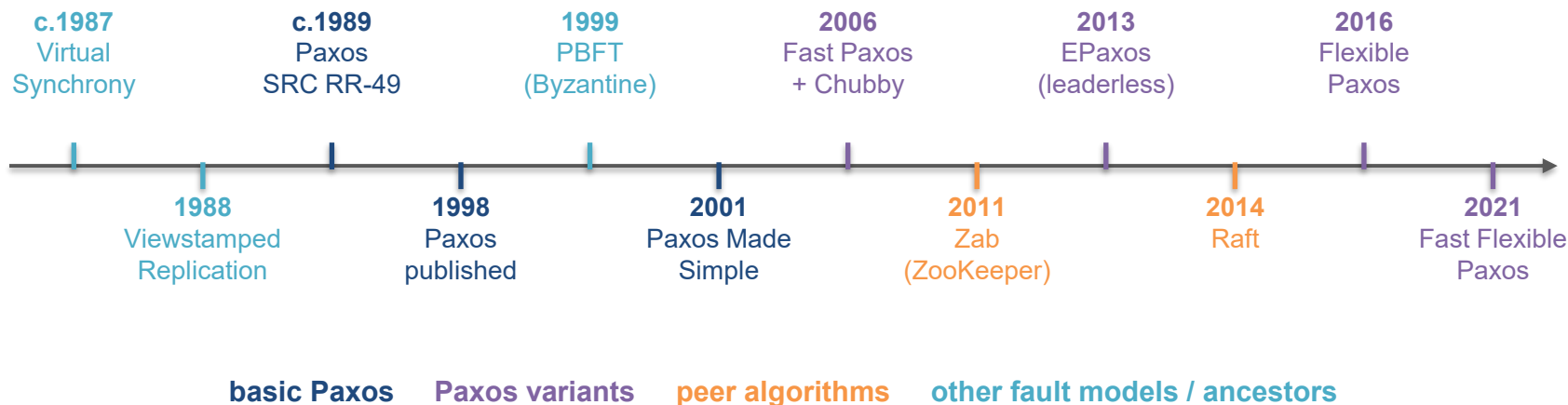
COMS4113

| Paxos |                                                                                                                                  | Raft                                                                                               |
|-------|----------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|
| +     | <b>The foundation.</b> Minimal assumptions; 35+ years of studied variants.                                                       | <b>Understandable by design</b> - and backed by a user study. Excellent reference implementations. |
| +     | <b>A design space.</b> Leaderless (EPaxos) and reconfigured quorums (Flexible) are possible <i>because</i> it assumes so little. | <b>One complete spec:</b> log replication, leader election and membership change together.         |
| -     | <b>"Paxos" names a dozen things</b> , and the paper-to-system gap is notoriously wide. Multi-Paxos was never fully specified.    | <b>The leader is a bottleneck</b> - all traffic through it, and its failure is a stall.            |
| -     | <b>Liveness needs engineering:</b> backoff and leases before duelling proposers are tamed.                                       | <b>Less room to manoeuvre.</b> No leaderless mode; the spec fixes choices Paxos leaves open.       |

**Same problem, same guarantees.** Paxos is the one to *reason* with; Raft is the one to *implement*.

# Re: Consensus is a family of protocols

"Paxos" names a **family**, not one algorithm - and it is not the only family. Here is the shape of it.



## You are here

The next ~25 slides are **basic Paxos** - the one you will implement. **Raft is not a Paxos variant**: it is a separate algorithm with the same guarantees. Both come at the end.

# Who Uses What in Production

| Paxos                                                                       |
|-----------------------------------------------------------------------------|
| <b>Google Chubby</b><br>the canonical Multi-Paxos paper, OSDI 2006          |
| <b>Google Spanner</b><br>2PC across Paxos groups                            |
| <b>Google Megastore</b><br>Paxos per entity group ( <i>extended Paxos</i> ) |
| <b>Azure Storage</b><br>Paxos in the Stream Manager ( <i>control path</i> ) |

| Raft                                                                    |
|-------------------------------------------------------------------------|
| <b>etcd</b><br>Kubernetes' datastore; textbook Raft                     |
| <b>CockroachDB</b><br>Raft per range ( <i>plus leaseholder reads</i> )  |
| <b>TiKV / TiDB ( transactional KV DB )</b><br>one Raft group per region |
| <b>Consul ( HashiCorp )</b><br>Raft for the service catalogue           |

## Two things people get wrong

**ZooKeeper is not in the Paxos column.** It uses **Zab**, and its own docs deny being Multi-Paxos.

**MongoDB is not in the Raft column.** Its pv1 elections are Raft-like; MongoDB's documentation never names Raft.

- Local Transactions
- 2PC
- Applications of consensus - and of 2PC - in real systems:
  - **Spanner**: Google's geo-distributed, fault-tolerant, scalable ACID database. **2PC across Paxos groups** - both protocols of the last two lectures, composed.
  - **Chubby**: Google's lock service, and the paper that showed what Multi-Paxos costs to build for real.
- Spanner also makes the bet Lecture 8's lease slide declined: it treats **bounded physical time** as a correctness primitive, not just an optimization.

# Key Papers: Paxos and Raft

- **[Lamport-1998]** Leslie Lamport. *The Part-Time Parliament*. ACM TOCS **16(2):133-169**, 1998. DOI 10.1145/279227.279229. Also SRC Research Report 49.
- **[Lamport-2001]** Leslie Lamport. *Paxos Made Simple*. ACM SIGACT News **32(4)**, 2001 - the accessible reformulation, **and the one to read first**.
- **[Lamport-2006]** Leslie Lamport. *Fast Paxos*. Distributed Computing **19:79-103**, 2006. DOI 10.1007/s00446-006-0005-x.
- **[Howard-2016]** Howard, Malkhi & Spiegelman. *Flexible Paxos: Quorum Intersection Revisited*. OPODIS 2016, **25:1-25:14**. arXiv 1608.06696.
- **[Howard-2021]** Howard, Charapko & Mortier. *Fast Flexible Paxos*. ICDCN 2021, pp. **186-190**. DOI 10.1145/3427796.3427815.
- **[Moraru-2013]** Moraru, Andersen & Kaminsky. *There Is More Consensus in Egalitarian Parliaments*. SOSP 2013, pp. **358-372**. DOI 10.1145/2517349.2517350.
- **[Ongaro-2014]** Ongaro & Ousterhout. *In Search of an Understandable Consensus Algorithm*. USENIX ATC 2014, pp. **305-319**. Fuller: Ongaro, *Consensus: Bridging Theory and Practice*, Stanford PhD thesis, 2014.

# Key Papers: the Wider Family

- **[Oki-1988]** Oki & Liskov. *Viewstamped Replication: A General Primary Copy*. PODC 1988, pp. **8-17**. DOI 10.1145/62546.62549 - predates Paxos.
- **[Junqueira-2011]** Junqueira, Reed & Serafini. *Zab: High-performance Broadcast for Primary-backup Systems*. DSN 2011, pp. **245-256**. DOI 10.1109/DSN.2011.5958223.
- **[Castro-2002]** Castro & Liskov. *Practical Byzantine Fault Tolerance and Proactive Recovery*. ACM TOCS **20(4):398-461**, 2002. (*The OSDI 1999 paper is the original.*) A different fault model - Lecture 14.
- **[vanRenesse-2015]** van Renesse, Schiper & Schneider. *Vive la Difference: Paxos vs. Viewstamped Replication vs. Zab*. IEEE TDSC **12(4):472-484**, 2015 - the best single source on how these relate.
- **[Burrows-2006]** Mike Burrows. *The Chubby Lock Service for Loosely-Coupled Distributed Systems*. OSDI 2006, pp. **335-350**.
- **[Chandra-2007]** Chandra, Griesemer & Redstone. *Paxos Made Live: An Engineering Perspective*. PODC 2007 - what it actually costs to build, and the standard citation for Multi-Paxos in practice.

*Multi-Paxos has no standalone paper: it is section 3 of the two Lamport papers, plus [Chandra-2007] for the engineering.*