Distributed Systems 1

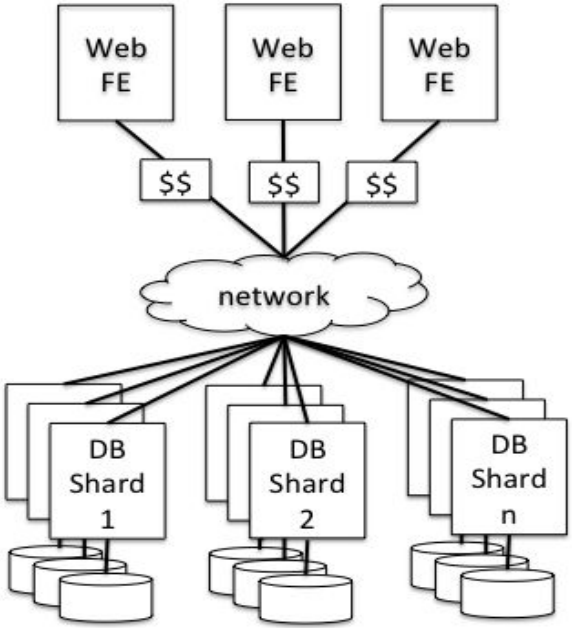
CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

Consensus Protocols
(Paxos)

Context

- We learned how to achieve atomicity, isolation in a sharded database.
- Today we learn how to achieve fault tolerance through replication. Problem of maintaining multiple replicated shards can ultimately be reduced to **consensus**.
- We discuss **Paxos**, the best known consensus protocol.

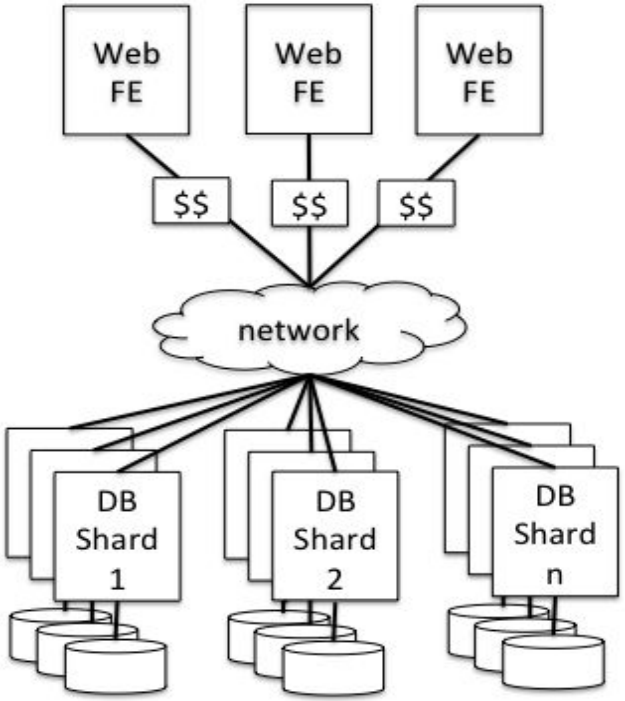


Outline

- Problem: Replicating ACID shards
- Mock protocol with 2PC
- Consensus protocols
- Paxos

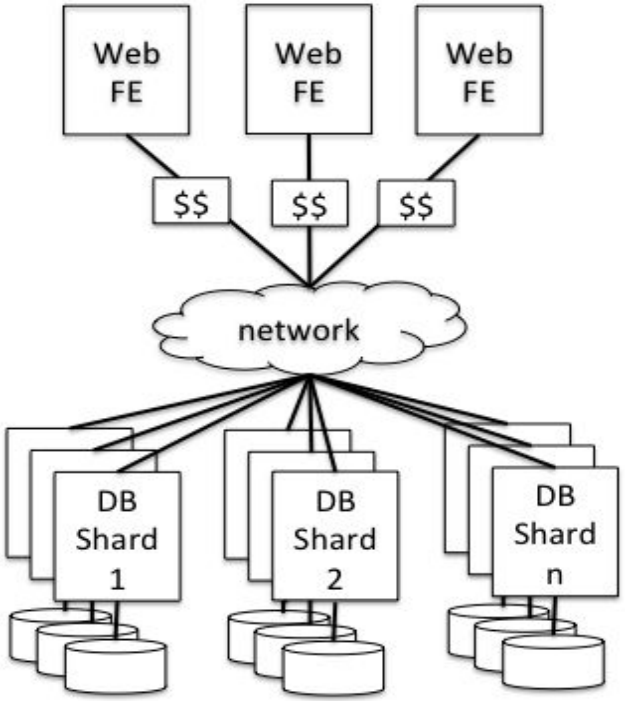
Problem: Replicating ACID shards

Example: Web Service with Transactions



- Assume: sharded database. Each DB shard runs an ACID engine (so runs 2PL+WAL). The shards coordinate via 2PC.
- Question: **Without shard replication**, what fault tolerance problems can arise?

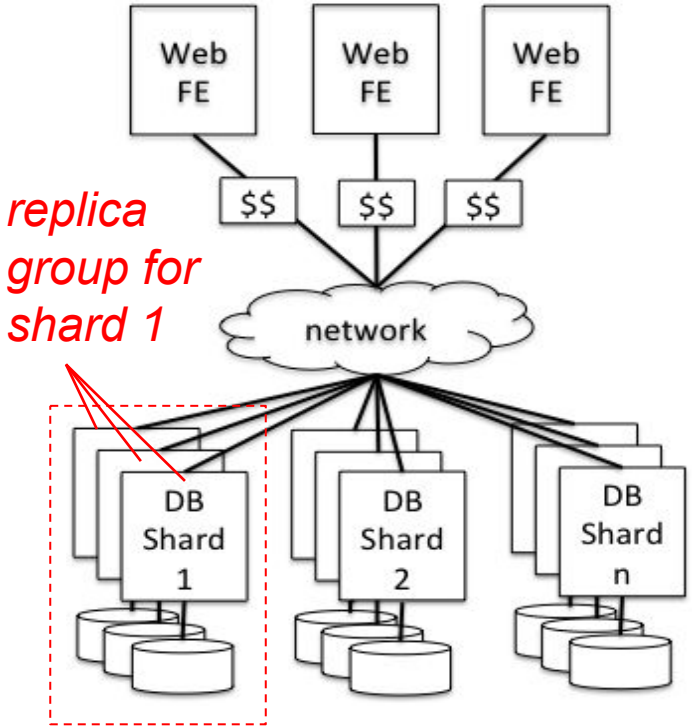
Example: Web Service with Transactions



Fault tolerance problems w/o replication:

- Data, WAL for each shard are stored on one disk. If disk dies, shard's data is lost. Durability problem!
- Even if disks don't fail, recall that 2PC can block if a shard server fails at inopportune time. Transactions interacting with the failed server block, along with many new transactions that transitively depend on rows locked by the blocked transactions. Availability problems!

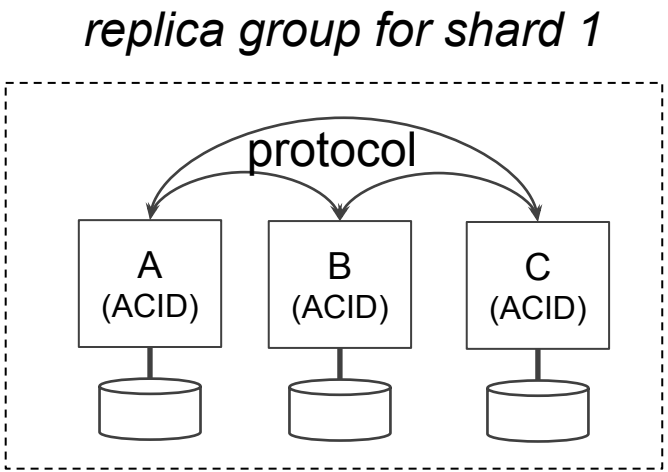
Solution: Replication



- Architecture:
  - Replicate each shard across multiple servers (each ACID, so they maintain WAL and do 2PL).
  - Replicas of a shard coordinate to maintain their state "in sync," ideally giving the illusion that they are a **single, (almost) always-on server**.
  - 2PC is executed across replica groups (we'll discuss how in future lectures). Because **replica groups** "never" die or become partitioned, 2PC "never" blocks.

Question 1: What State to Replicate?

- Disk image?
- In-memory image?
- WAL?
- Locks?
- … Anything else?



Basic Answer: Replicate WAL

- Claim: If all replicas execute **all WAL ops**, **in the same order**, then all other state (DB image, locks, …) will be reconstructed in the same way across replicas (assuming deterministic operation).
- It *can* be useful to be able to push checkpoints of the DB to a recovering/new replica, but we'll ignore that for now and focus on replicating the WAL.

Question 2: What Semantic to Require?

- Requirement: **all replicas apply (1) the same log entries, (2) in the same order**.
- Otherwise, inconsistencies can occur.
- As examples, consider:
  - One replica skips log entry for an update while others apply it.
  - One replica receives two updates for a particular row in one order while another receives them reversed.

Question 3: How to Replicate?

- Requirement: all replicas apply (1) the same log entries, (2) in the same order.

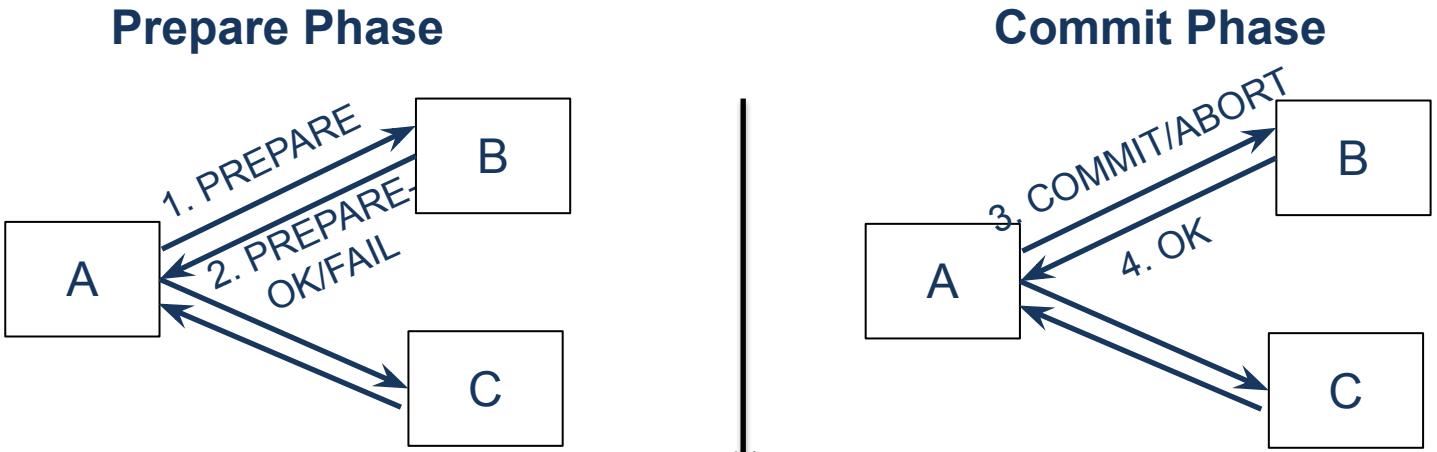
- One idea: **2PC**.
  - 2PC ensures that all participants either do all ops or don't do any of the operations.
  - Could we use this protocol for WAL replication?

Mock protocol based on 2PC

Mock 2PC-based Replication

BREAKOUT ACTIVITY!

- A, B, C are replicas of a single shard. They need to coordinate to apply all WAL entries in the same order.
- Discuss how might it work and what problems would arise.



Roxana's Mock

- One replica assigned as TC. TC decides on order of ops in the log and performs 2PC for each log entry, every time blocking for the protocol to finish before launching a 2PC for the next log entry.
- This ensures that all replicas:
  - Apply all log entries (thanks to 2PC).
  - Apply log entries in the same order (thanks to sequential way in which TC performs log entry pushes to participants).
- Can be optimized to do 2PC for batches of log entries -- **when** would you need to push a batch???

Problems with Mock

1. **NOT fault tolerant (but durable):**
   - Because TC must wait for all replicas to reply that they are going to perform the update, the coordinator needs to block every time one replica is slow, disconnected, or dead.
   - But the mock does provide more durability than 2PC across shards.
2. When the **coordinator dies**, someone else must become coordinator. Yet, we must have only one (at most) coordinator, otherwise different coordinators may impose different orders on log entries. This is called **leader election** and is not addressed in 2PC, which assumes a static coordinator!

Consensus protocols

Consensus Protocols

- Require only a **majority of nodes** to be up at any time in order to make progress.
- Similar to 2PC, but instead of waiting for all participants to respond, they wait for a majority of the replicas to respond.
  - In a **fail-stop failure model** (i.e., nodes are not malicious), the majority needed is a **simple majority**; i.e., one can tolerate f simultaneous failures with 2f+1 replicas.
  - In a **malicious failure model**, one needs a **super-majority**, i.e., one can tolerate f simultaneous failures with 3f+1 replicas.

Simple Majorities

- **There cannot exist two majorities in a given group at the same time.**
  - This means that if a node obtains OKs from a majority of nodes – say in a first phase like 2PC's – then another node (e.g., another simultaneous coordinator) is guaranteed to not have obtained OKs from a majority of the nodes.
  - This lets us replace a dead Coordinator with a new one without introducing inconsistencies. That's how we address the leader election problem.
- **Any two majorities of a group will overlap in at least one node.**
  - This means that if an old Coordinator obtained OKs from a majority of the nodes, then sent COMMIT messages that were received by a majority of the nodes, and subsequently crashed before it could inform the other nodes of the COMMIT outcome, then a new Coordinator that is "elected" subsequently, will learn about the outcome by talking to any (other) majority, and so it can continue the commit process that the first (now dead) Coordinator began.

Paxos and RAFT

- **Paxos** [Lamport-1998]: Original protocol. Solves the **basic consensus problem** as defined in the Agreement lecture (consensus on the **value of a write-once register**, with the consistency, validity, and termination requirements).
- **RAFT** [Ongaro-Ousterhout-2014]: More recent, operates at a higher level of abstraction, and shows very clearly how to replicate the WAL (for example) to implement fault-tolerant transactions.
- We'll focus on Paxos, and its extension to WAL.

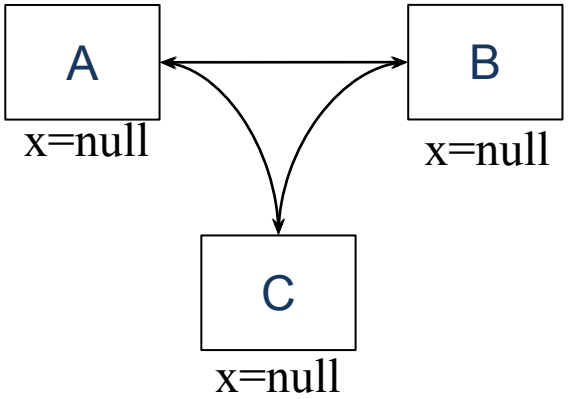
Paxos

Paxos

- Widely used in industry to solve various instances of consensus that occur in DS:
  - Google: Chubby (Paxos-based distributed lock service), Spanner (transactional storage)
  - Yahoo: Zookeeper (Paxos-based distributed lock service)
  - Open source: libpaxos (Paxos-based atomic broadcast)

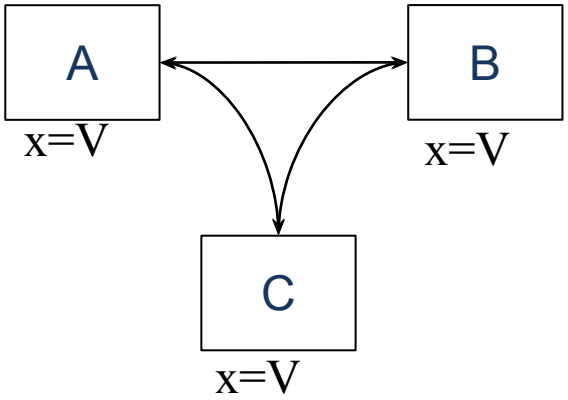
Gist

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



Gist

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



Gist

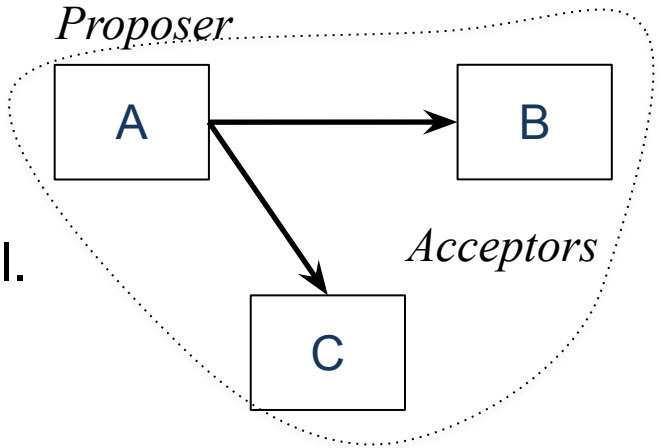
- Paxos solves the generic problem of consensus: **N nodes want to agree on the value of a *write-once register*.**

- The protocol guarantees:
  1. **consistency** (all non-faulty nodes choose the same value);
  2. **validity** (the chosen value was proposed by a proposer).

- The protocol is likely to achieve but does not guarantee:
  3. **termination** (eventually, a value is chosen).

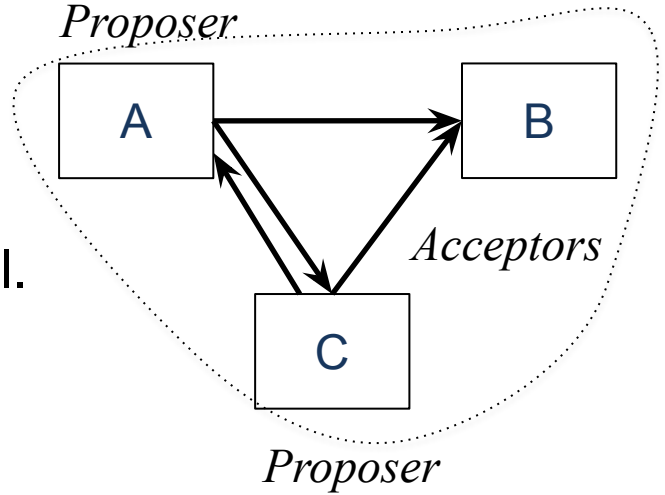
Gist

- All nodes can fulfill **two roles**:
  - *Proposers*: issue a series of rounds of proposals, ordered with logical clocks.
  - *Acceptors*: accept or reject values from proposers according to a specific protocol.



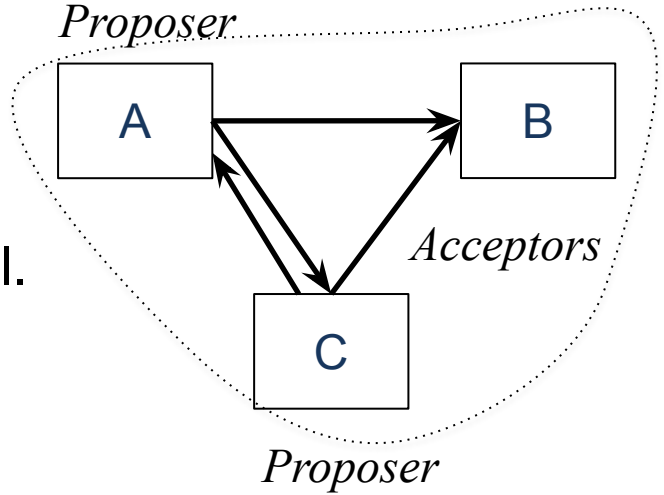
Gist

- All nodes can fulfill **two roles**:
  - *Proposers*: issue a series of rounds of proposals, ordered with logical clocks.
  - *Acceptors*: accept or reject values from proposers according to a specific protocol.



Gist

- All nodes can fulfill **two roles**:
  - *Proposers*: issue a series of rounds of proposals, ordered with logical clocks.
  - *Acceptors*: accept or reject values from proposers according to a specific protocol.
- A value is *chosen* when a majority of acceptors have accepted it.
- A proposer announces a chosen value or tries again if it's failed to converge on a value.



The Protocol

- State maintained by each node:
  - Np: highest proposal number seen to date (initially nil);
  - Na: highest accepted proposal (initially nil);
  - Va: the value of the highest accepted proposal (initially nil);
  - Done: whether consensus has been reached (initially false).
- Protocol has three phases:
  - Propose
  - Accept
  - Decide

Phase 1: Propose

- @Proposer: (assumes Done=false)
  - Choose a new proposal number, N > Np.
  - Send <PROPOSE, N> to acceptors (including himself).
  - Wait until a majority of acceptors return PROPOSE-OK. If time out, back off and restart Paxos.

- @Acceptor: Upon receiving a <PROPOSE, N> request:
  - If N > Np then:
    - Update Np = N
    - Reply <PROPOSE-OK, Na, Va>

Phase 2: Accept

- @Proposer: (assumes PROPOSE-OKs from majority acceptors)
  - Choose V := the value of the highest-numbered proposal among those returned by the acceptors (or any value if no Va returned).
  - Send <ACCEPT, N, V> to all acceptors (including himself).
  - Wait until a majority of acceptors return ACCEPT-OK. If time out, back off and restart Paxos.
- @Acceptor: Upon receiving an <ACCEPT, N, V>:
  - If N >= Np then:
    - Update Np = N, Na = N, Va = V
    - Reply <ACCEPT-OK>

Phase 3: Decide

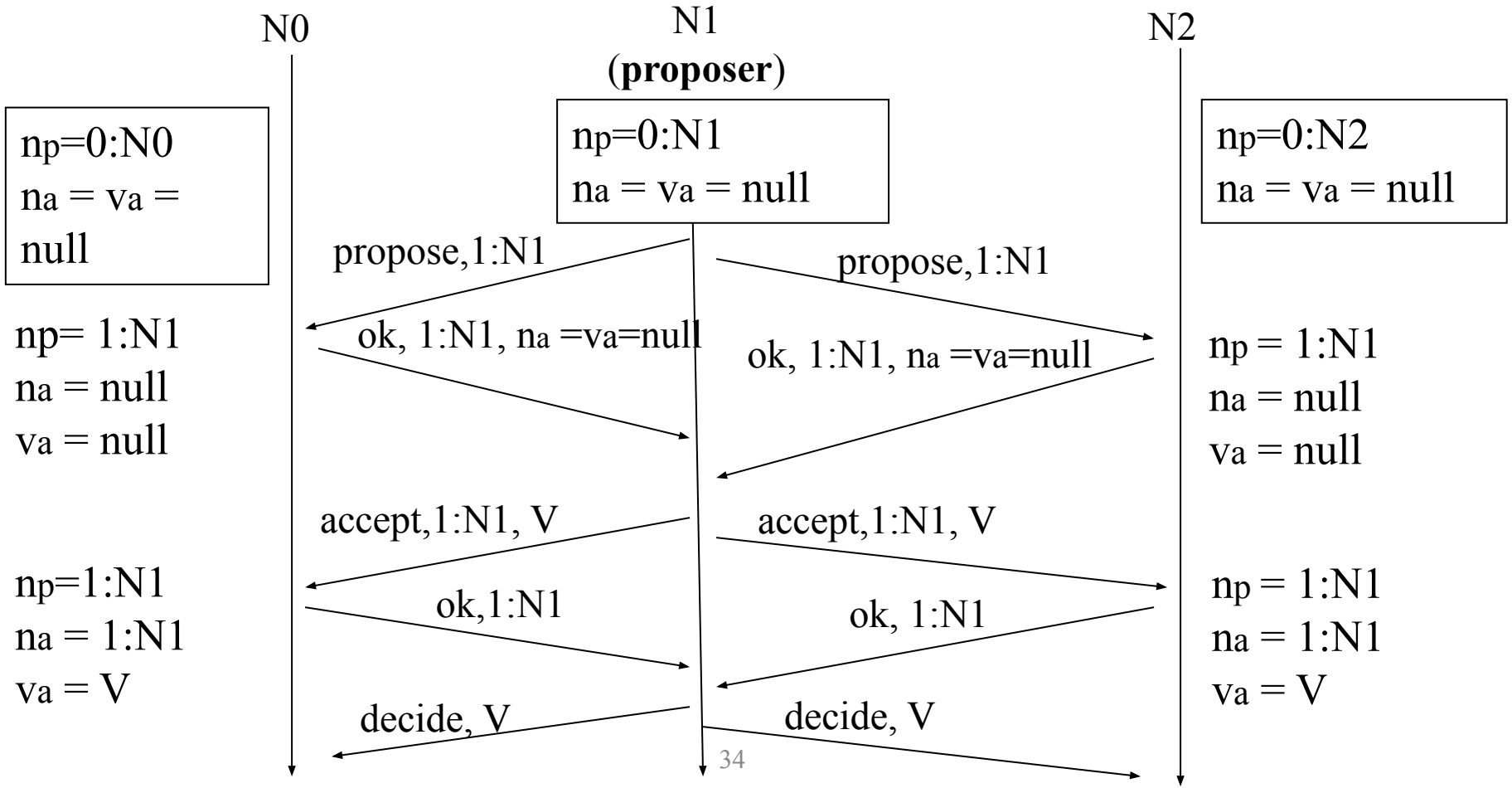
- @Proposer: (assumes ACCEPT-OKs from majority of acceptors):
  - Send Done to client, signaling that consensus has been reached.
  - Send <DECIDE, N, V> to all acceptors (including himself). [Can keep resending until all reply, but realize that acceptors can learn decision from other proposers too.]

- @Acceptor: Upon receiving a <DECIDE, N, V>
  - If N >= Np then:
    - Set Np = N, Na = N, Va = V
    - Reply <DECIDE-OK>
    - Set Done = true and terminate Paxos.

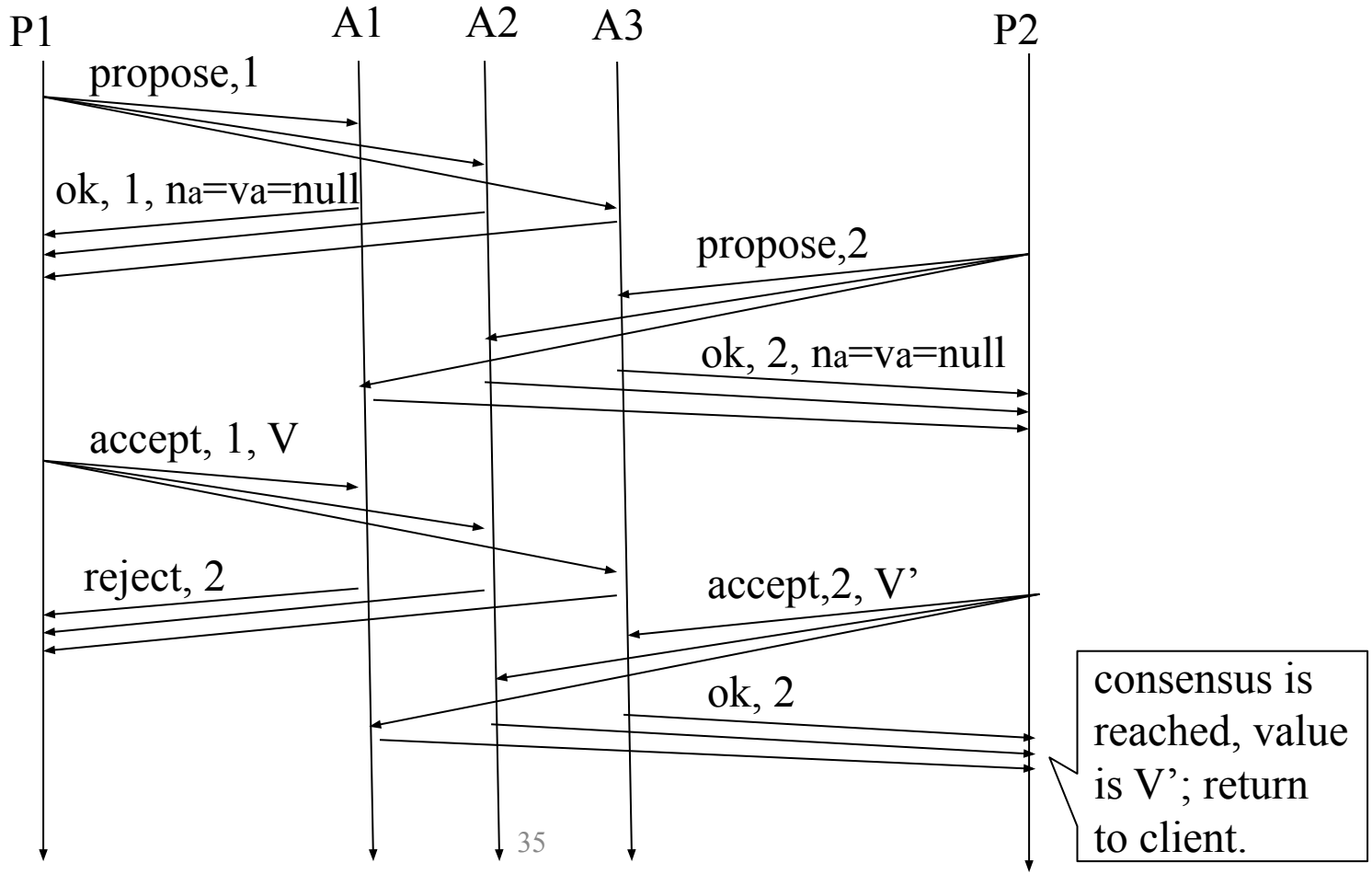
Examples

- Paxos is best understood by first reading protocol, then examples, then reading protocol, then examples, …
- We'll go through several examples next.
- Please re-read protocol at home and construct your own examples, questioning the protocol. It's the best way to understand it!

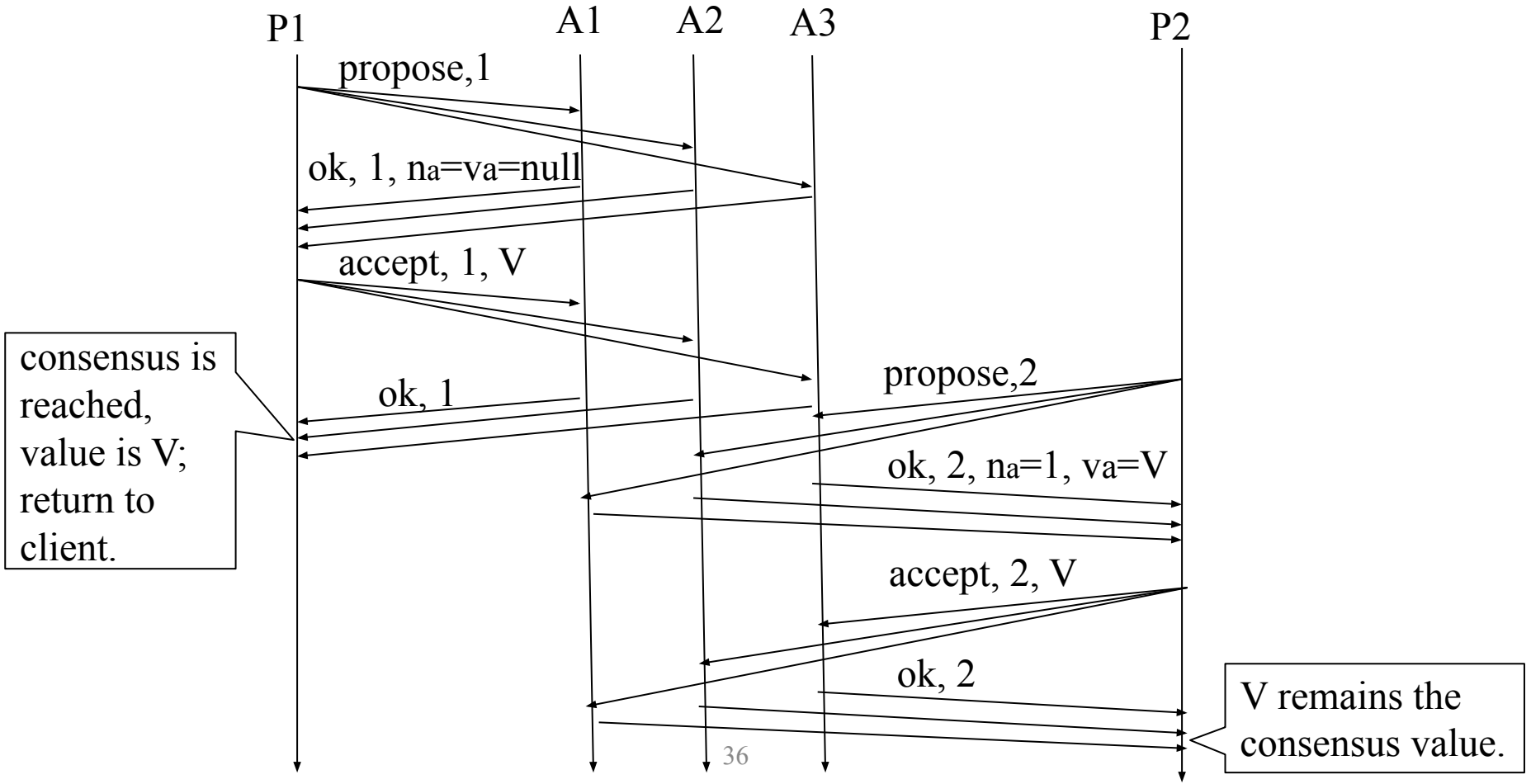
1. Single Proposer



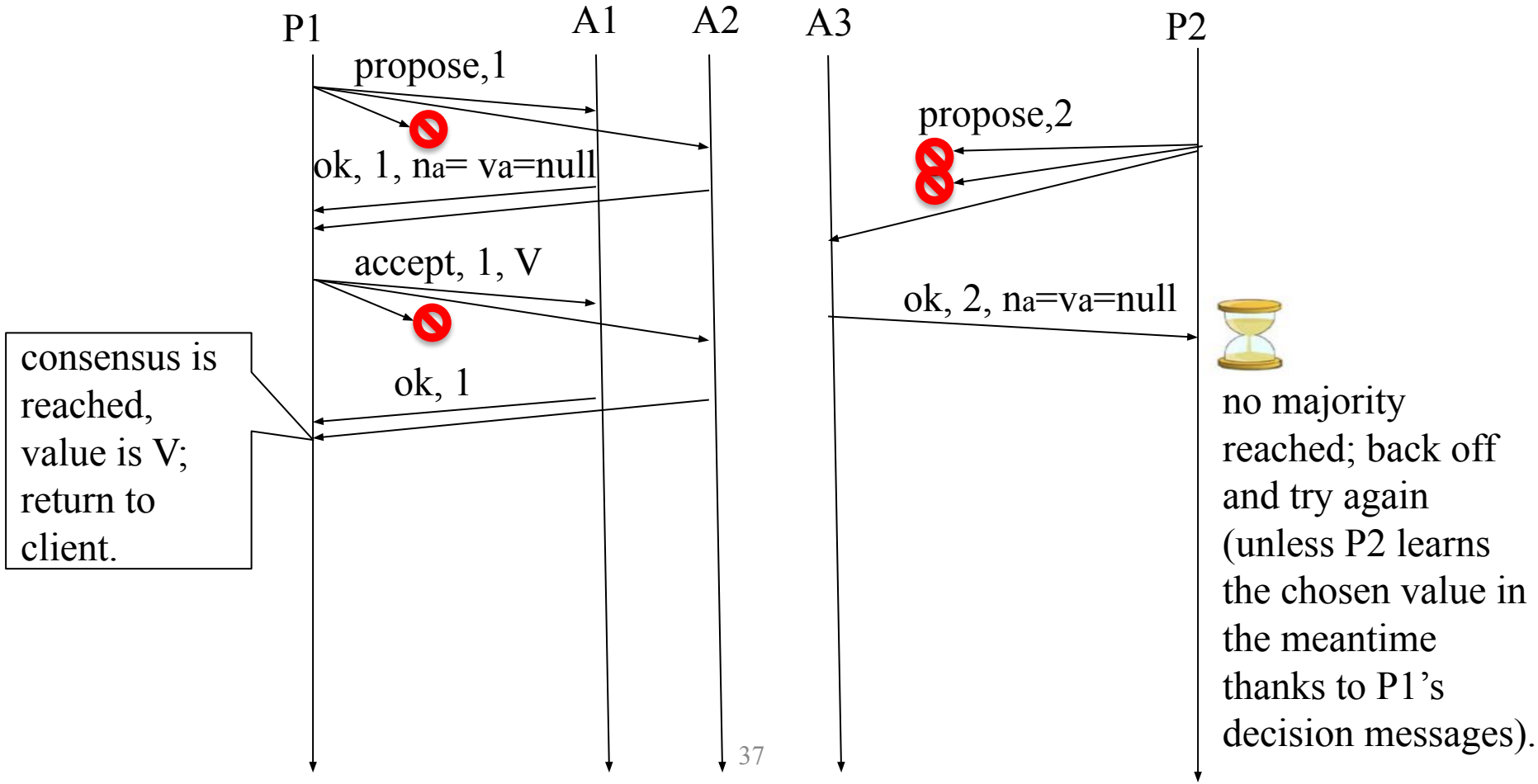
2. With Concurrent Proposers



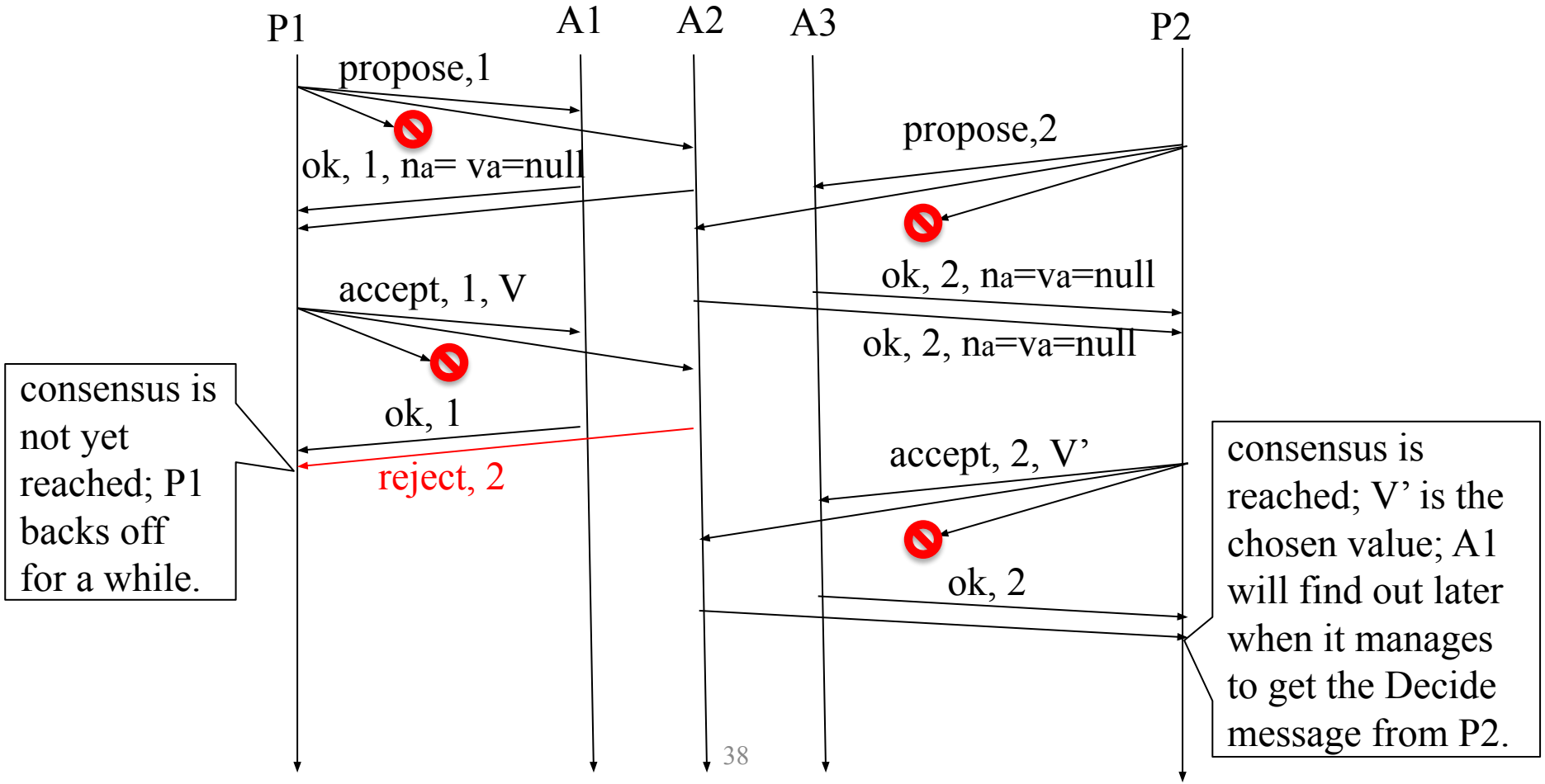
3. With Sequential Proposers



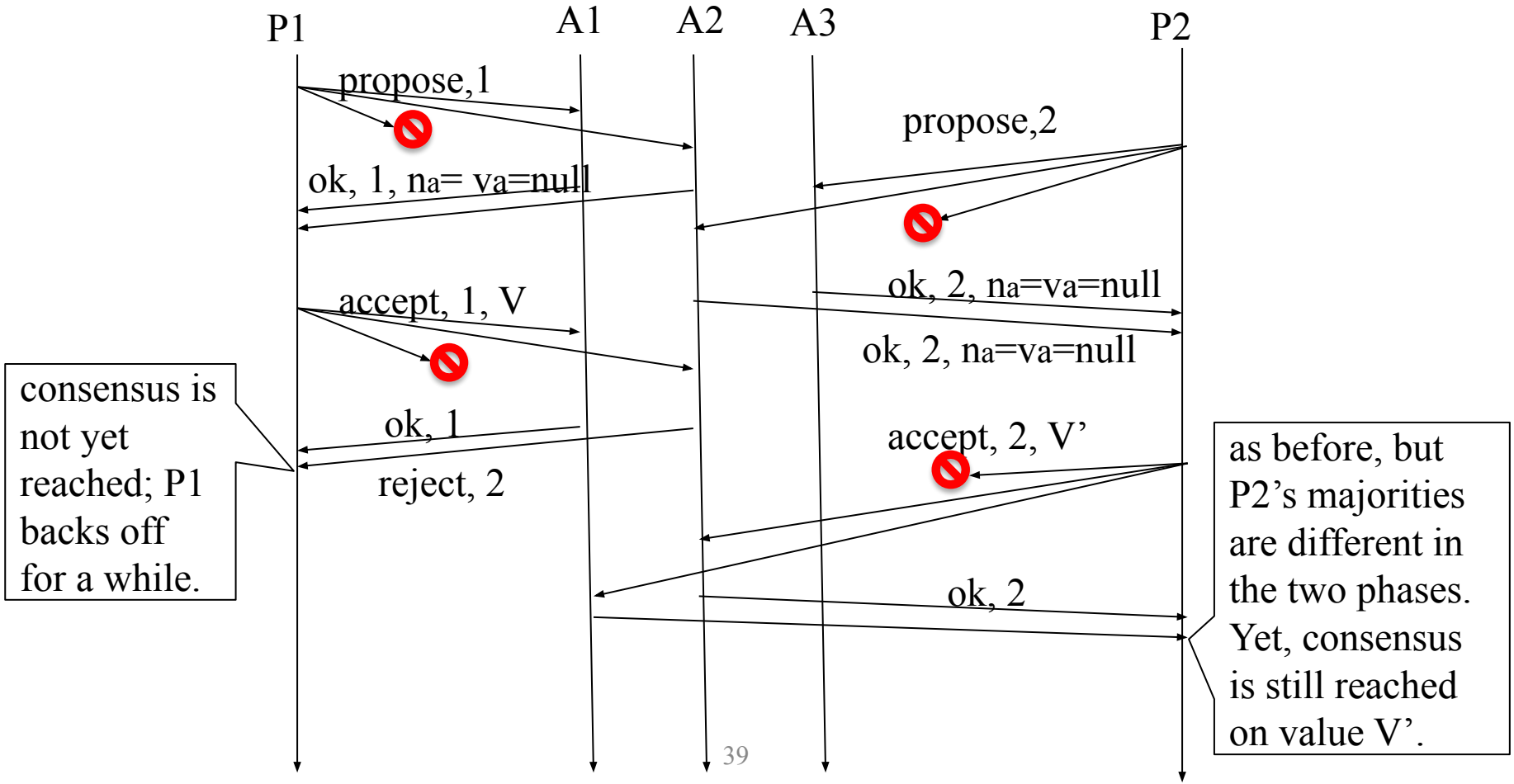
4. With Failures



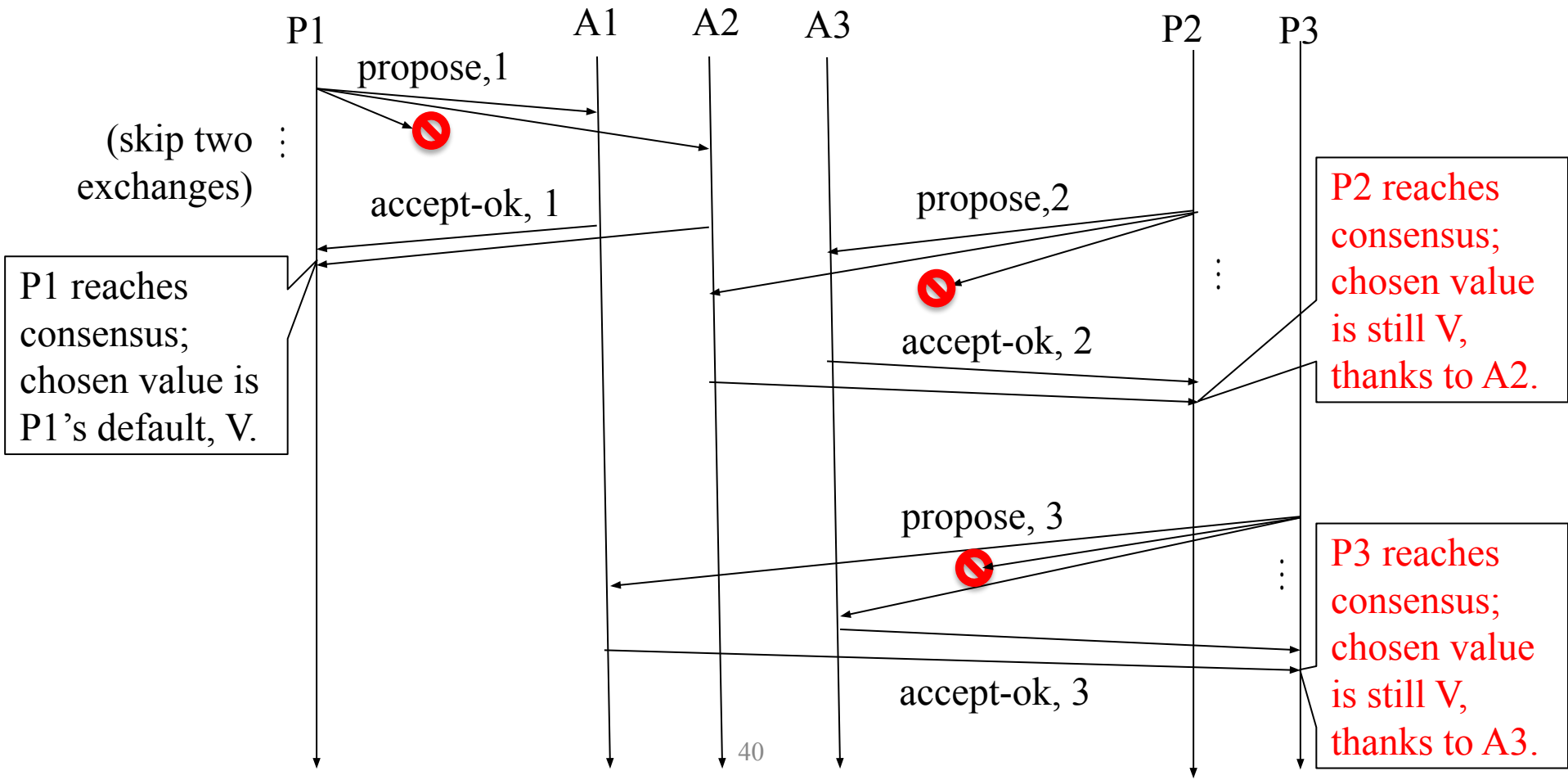
5. With Failures



6. With Failures



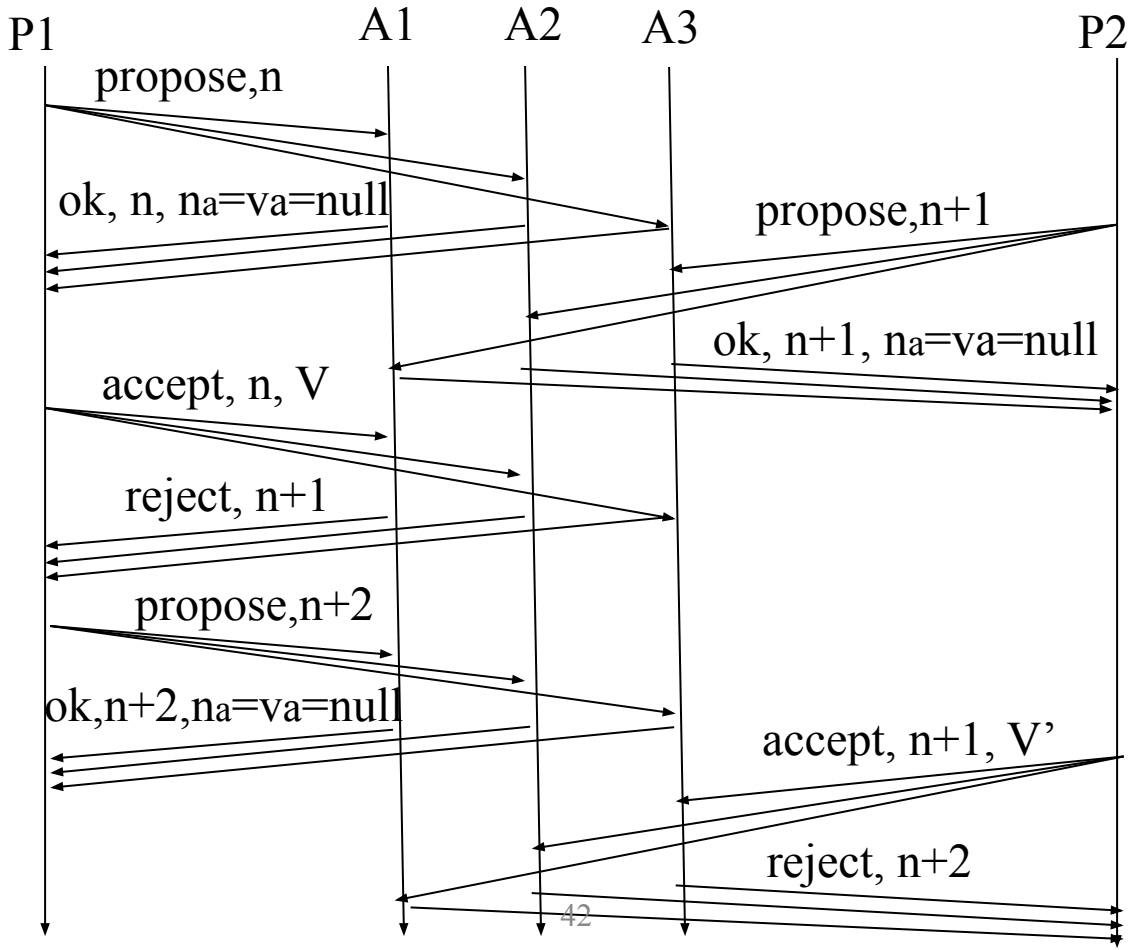
7. With Failures



Paxos Is Fault-Tolerant

- A Paxos *run* consists of one or more rounds conducted by different proposers. New proposers continue the work of previous proposers.
- If one proposer dies, another one times out and offers to be the proposer. Because of how the protocol is structured (that value choice based on highest-Na), the new proposer will continue propagating a formerly chosen value.
- A Paxos run is successful if at least a majority of the nodes is up and accepts the proposal.
- But, there are degenerate cases where **Paxos doesn't finish** (next slide). These can be made unlikely w/ random back-offs.

Paxos May Not Terminate



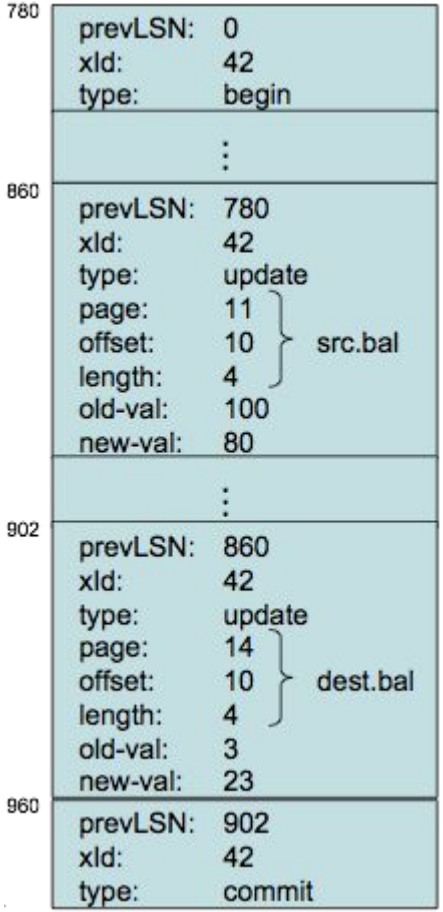
Dueling proposers.
Solution: random backoff.

Extending to Read-Write: Multi-Paxos

- Preceding protocol solves the basic problem of consensus on write-once registers.
- Real consensus problems don't look like that because real data structures one wishes to replicate are rarely write-once.
- But many useful data structures can be reduced to a set of write-once registers.
- Example: The WAL is a long vector of write-once registers.
  - Each index in the log (record #1, #2, …) is written once and never updated.

Example: Replicating WAL

- You start out with an empty log, i.e., no value is assigned to any of the indexes.
- For each index, you use Paxos to ensure the replicas agree on what value (record, such as begin, update, commit, prepare,…) is recorded at a particular index in the log.
- When a replica has a record to append to the log, it picks an unused index and proposes to become a proposer. If no acceptor returns a value for that index, the proposer can use its record as the value.



Inefficiencies of Multi-Paxos

- Preceding description of multi-paxos is inefficient:
  - Clients are allowed to interact with any replica, so proposers may duel, causes a lot of useless work.
  - Each time a replica becomes a proposer, it needs to run the first phase (PREPARE) to get agreement on N.
- The solution is to have a **stable Leader**, similar to 2PC except the leader can change seamlessly if needed.

Leader-based Multi-Paxos

- Say proposer P completes first phase. Then, P can assume the role of a **Leader** for a predefined period of time (**lease time**).
- Any replica that is not a leader rejects client requests and forwards the client to the node it believes is the leader.
- P interacts with clients and fast-tracks its proposals from the ACCEPT phase.
- Changing the Leader is achieved with leaderless multi-Paxos, with versions of the leader being indexed by a "view number."
  - A replica will reject a Leader change until it hasn't heard from former leader for a predefined amount of time (**lease time**).

Lease

- Gives a node permission to act in a certain role for a period of time (*real time*!), with the possibility of renewal in that timeframe.
- Used throughout DSes as an optimization.
  - Logical clocks (Np, Na in Paxos) are used for safety/correctness, as they don't raise synchronization challenges.
  - Physical time, through leases, is used for optimization.
- Assumption: **small clock skew** during lease period!
  - This is why leases can't be too long!
  - E.g., 30-second leases are typical.

Next Time

- Applications of Paxos (and 2PC) in real life:
  - Spanner: Google's geo-distributed, fault-tolerant, scalable ACID database.
  - Chubby: Google's lock service.

Key Papers

- [Lamport-1998] Leslie Lamport. *The Part-time Parliament.* In *ACM Transactions on Computer Systems*, 1998.
- [Lamport-2001] Leslie Lamport. *Paxos Made Simple.* In *ACM SIGACT News*, 2001.
- [Ongaro-Ousterhout-2014] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm.* In *USENIX ATC, 2014.*