CONSENSUS PROTOCOLS
Or: How to Implement Fault Tolerant Transactions

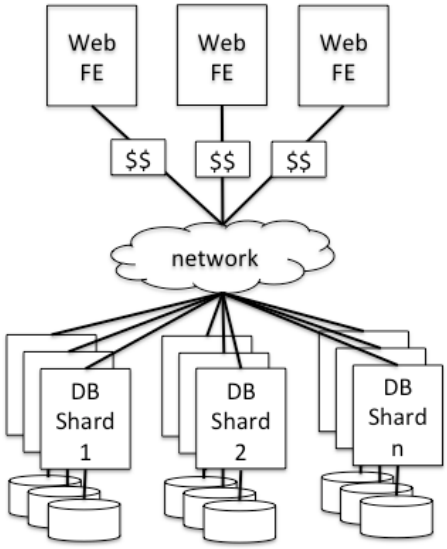
Last time we talked about *sharding* – a key mechanism for scaling distributed systems – and showed how to implement transactions on sharded databases with two-phase commit (2PC), an atomic commitment protocol. Today we'll focus on *replication* – a key mechanism for fault tolerance in distributed systems. We will show that, conceptually, the problem of maintaining multiple replicated shards can be reduced to an instance of the consensus problem. We will discuss one particular protocol for consensus, Paxos, and provide references for a second one, RAFT. Next time we'll put the 2PC and Paxos together by looking at Spanner, Google's **fault tolerant and scalable ACID system**.

I. SEMANTIC CHALLENGES RAISED BY REPLICATION

Suppose we have a sharded DB that supports the notion of transactions. Individual shards handle different portions of the database, using locking and write-ahead logging to implement transactions within each shard, and coordinating via 2PC to preserve atomicity of cross-shard transactions. From a *fault tolerance* perspective, there are two issues:

1. The data and write-ahead log of each shard are stored on one disk. If that disk dies, the shard's data is lost. That's a *durability* problem (which we assumed away last time).
2. Even if disks do not fail, if one shard server crashes, any transaction that needs access to that shard's data will have to block waiting for it to come back. Worse, if at the time the server crashed there were some ongoing cross-shard transactions that involved that server, then those transactions may have to wait for the server to come back before it can complete. Moreover, any *new* transaction that accesses some common rows as the blocked transactions will also have to block waiting for the transaction to finish. Those are serious *availability* problems.

The way we handle these problems is using **replication**: let's replicate each shard of the database. Take a look at the architecture to the side, which illustrates exactly this setting. The goal is to replicate enough of the information maintained by each shard server so another server can "take over" upon the failure of a particular shard server.



Question: For each shard, what information should we replicate?
- The rows of each shard?
- The write-ahead logs?
- The locks being kept on behalf of ongoing transactions?
- Anything else?

Answer: We should at least replicate **the write-ahead logs**. The database rows (i.e., the content of each shard) can be obtained by just applying the log entries against some checkpoint of the database, so we don't have to replicate them. Locks may be useful to replicate, however remark that if a failure occurs, the server that takes over could just abort any ongoing transactions it sees in the log.

So, for this class, let's focus on replicating just the write-ahead log for each shard. And from now on, let's focus on the single-shard case: i.e., we don't have sharding anymore, we only have one database and we aim to maintain its **write-ahead log** (a.k.a., **transaction log**) on some number of replicas.

Question: What semantic challenges arise when replicating a transaction log?
Answer: We need to make sure that all replicas see all log entries, in the same order. Otherwise, inconsistencies can arise:

- Example 1: Suppose a replica skips the log entry for a particular update while the other replicas see it. This means that the first replica will not perform the update against its version of the database while the other replicas will. This means that the replicas' views of the database's state will diverge.
- Example 2: Suppose a replica receives two updates for a particular row (potentially coming from different transactions) in a particular order, and another replica receives those same updates, but in the reverse order. This means that the two replicas' views of the database's state will diverge.

These are both examples of consistency challenges raised by replication.

II. WHY 2PC CANNOT ADDRESS THESE CHALLENGES

On the face of it, 2PC might appear to address the consistency challenges raised by replication (or at least it might appear to be a good starting point for a solution). 2PC seeks to make sure that all participants of a distributed system perform a certain operation, or none of them do. Why couldn't we use the same protocol to make sure that, upon an update, all replicas either apply a particular update or none of them do (i.e., that update fails, and hence presumably the whole transaction fails because of that)? Seems like it might work…

Specific proposed approach:

- Let's use the 2PC architecture: one replica is the Coordinator, the other replicas are the "participants." The Coordinator receives all client requests (reads, writes, and commits to the DB as part of various transactions) and executes them the way we described in the previous class.
- To add a log entry to the log, a Coordinator performs 2PC with the other replicas to make sure that they also commit the log entry into their logs.
- This ensures that (1) all replicas will see all log entries (thanks to 2PC) and (2) all replicas will see the log entries in the same order (thanks to the sequential way in which the coordinator performs these log entry pushes to the participants).
- The approach may be optimized to do 2PC for a batch of log entries, only upon the commit of a transaction, which is when we would normally sync the log to disk anyway in a transactional system (see WAL slides from the previous lecture).

Problems with the proposed approach:

1. NOT fault tolerant: because the Coordinator must wait for all replicas to reply that they are going to perform the update, if any of these replicas fail during one of these 2PC protocols, the Coordinator needs to block waiting for the replica to come back. We're back to the availability issues described at the beginning of Section I, so despite using 2PC for fault tolerance, we did NOT get a fault tolerant system (though maybe we addressed the durability issues).
2. When the Coordinator dies, someone else needs to take over as Coordinator. Yet, we need to make sure that only (at most) one node acts as Coordinator at any given time. Otherwise, different

coordinators might impose different orders on concurrent updates/transactions, which may lead back to inconsistencies. Having the replicas "agree" on who's the Coordinator is called the "leader election," and it's a common problem in distributed systems.

Addressing the preceding two problems requires a complete re-think of 2PC. If we want to tolerate some failures, we'd better have some protocol that makes progress when only a subset of the nodes is up. But how many replicas need to be up to tolerate a certain number of failures? And how should they coordinate to make progress despite these failures, which can occur at any time during the protocol's execution? The answer comes from what are called "*consensus protocols*." These differ from *commitment protocols* (such as 2PC) in that they require a fraction of the nodes to be up. And that fraction, turns out, needs to be a *majority* of the replicas.

III. CONSENSUS PROTOCOLS

Consensus protocols require only a **majority of nodes** to be up at any time in order to make progress. A simple way to imagine these protocols is that they are similar to 2PC, but instead of waiting for all participants to respond – as 2PC does after the Coordinator sends PREPARE messages to the participants – they wait for a majority of the replicas to respond. In a crash-only 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. We'll focus on the *crash-only model* in this class.

Properties of **simple majorities (a.k.a., majorities)**:

1. **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 as 2PC's – then another node (e.g., another simultaneous coordinator) is guaranteed to not have obtained OKs from a majority of the nodes. This allows us to replace a dead Coordinator with a new one without introducing inconsistencies. That's how we address the leader election problem.
2. **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.

We'll look at two consensus protocols: Paxos and RAFT. Paxos is the original protocol introduced by Leslie Lamport in 1998. The 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, see https://columbia.github.io/ds1-class/lectures/04-agreement-problem.pdf). The protocol is thus fairly low-level in terms of the problem it addresses; extensions are needed (and do exist) to solve bigger problems, such as replicating the WAL. The RAFT protocol is 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 discuss the basic Paxos protocol first, then we'll talk about an extension to Paxos, called *multi-Paxos,* needed to replicate a WAL. Then you'll watch a talk about RAFT at home. Multi-Paxos and RAFT are subjects of the final quiz.

IV. PAXOS

Paxos is widely used in industry to address many instances of the consensus problems and provide valuable programming abstractions for distributed systems:

- 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:

- The problem Paxos solves: **N nodes want to agree on the value of a write-once register.**
- There are two types of nodes at any time, and any node can be of any type:
  - *Proposers*: issue a series of rounds of proposals, ordered with logical clocks.
  - *Acceptors*: accept or reject values they receive from the proposers according to a particular protocol.
- A value is *chosen* (a.k.a., consensus is reached) 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 guarantees (1) *consistency* (all non-faulty nodes choose the same value) and (2) *validity* (the chosen value was proposed by a proposer). It ensures (3) *termination* (eventually, a value is chosen) with high probability but does not guarantee it.

The Protocol

A Paxos **round** starts when a node decides to propose something. This may happen because it's received some external request, or because it's trying to finalize a previously unresolved proposal.

Phase 1 (Propose)

Proposer steps:

- Chooses a new proposal number, N
- Sends <PROPOSE, N> to acceptors (includes himself)
- Waits until a majority of acceptors return PROPOSE-OK responses
- If time out waiting, back off then restart Paxos

Acceptor steps:

// Acceptors maintain some state:
// Na: highest accepted proposal
// Va: the value of the highest accepted proposal
// Np: highest proposal number they've seen to date.

- Upon receiving a <PROPOSE, N> request:
  - If N > Np then
    - update Np = N

        - return <PROPOSE-OK, Na, Va>
    - Else: return <REJECT, Np>

Phase 2 (Accept)

Proposer steps:

- If proposer gets PROPOSE-OK responses from a majority of acceptors:
    - Choose V = the value of the highest-numbered proposal among those returned by the acceptors (or any value he wants if no Va was returned by any acceptor)
    - Send <ACCEPT, N, V> to all nodes (acceptors)
    - Wait until a majority of acceptors returns ACCEPT-OK response
- Else, or if time out waiting, then back off then restart Paxos

Acceptor steps:

- Upon receiving an <ACCEPT, N, V>:
    - If N >= Np then:
        - update Np = N, Na = N, Va = V
        - return <ACCEPT-OK, N>
    - Else: return <REJECT, Np>

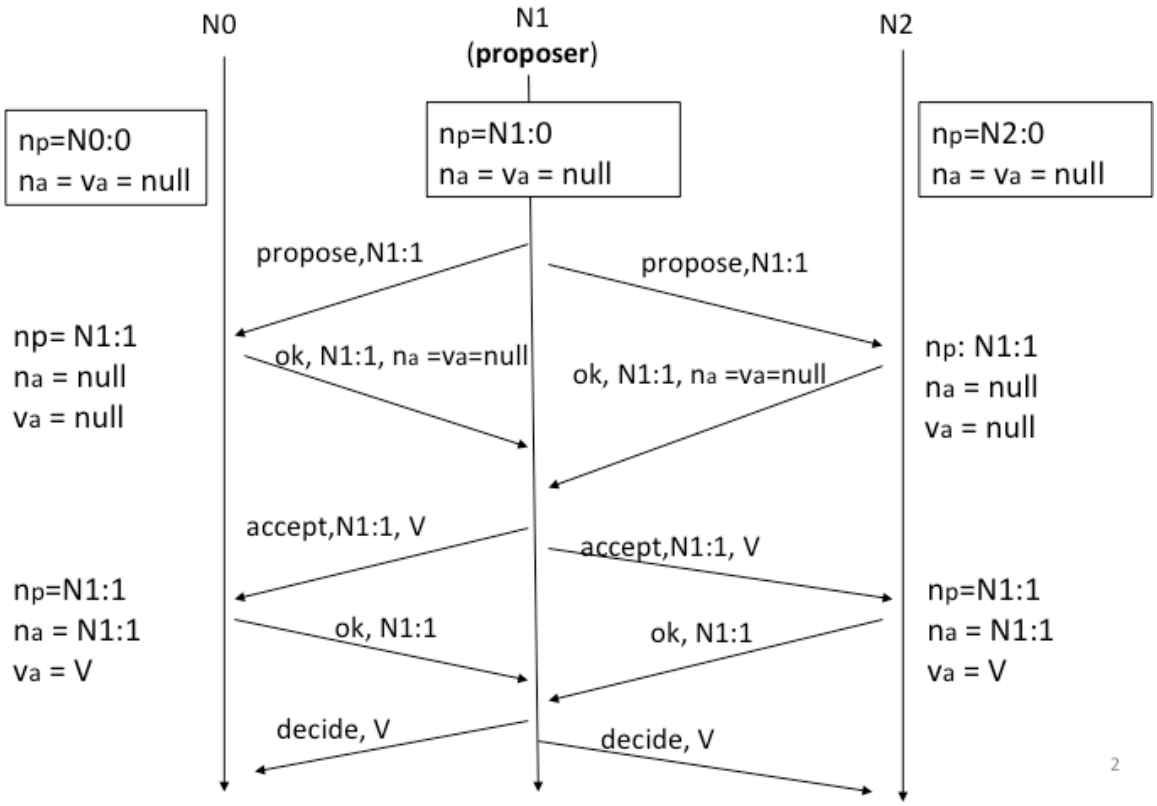
Phase 3 (Decide)

Proposer steps:

- If proposer gets ACCEPT-OK responses from a majority of acceptors:
    - Return Done to client, signaling that consensus has been reached. // Client can now: enter a critical section, start behaving as a leader, etc. – whatever the value signified to him.
    - Send <DECIDE, Va> to all replicas, and keep sending until you get DECIDE-OKs from everyone. // This phase is so that nodes close the protocol, and so that nodes that might not have heard previous ACCEPT messages learn the chosen value.
- Else back off then restart Paxos

Example Functioning

Paxos can be hard to understand. Below is an example of its basic functioning when no failures or concurrent proposers are involved.



We provide examples of Paxos functioning with failures and concurrent proposers here: https://columbia.github.io/ds1-class/lectures/07-paxos-functioning-slides.pdf. You are strongly encouraged to come up with your own examples that "challenge" the protocol to see how and why it works. That's the best way to internalize the protocol.

Some examples of useful questions to think about (akin to the ones that might appear on exam):

- When is value V chosen?
- When proposer receives a majority prepare-ok and proposes V
- When a majority nodes accept V
- When the proposer receives a majority accept-ok for value V
- What if more than one proposer is active?
- Suppose two proposers use different proposal numbers, N0:10, N1:11. Can both proposers see a majority of prepare-ok?
- What if proposer fails while sending accept?
- What if a node fails after receiving accept?
    - If it doesn't restart
    - If it reboots
- What if a node fails after sending prepare-ok?
    - If it reboots

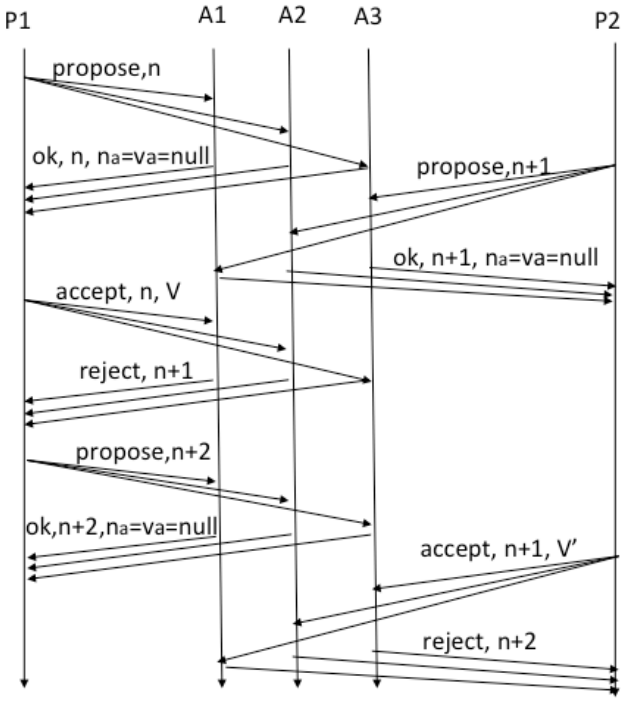
Paxos Properties

Paxos can be shown to meet the consistency (a.k.a. safety) and validity requirements of consensus. It is not guaranteed to meet the termination (a.k.a. liveness) requirement. Indeed, recall the FLP result of impossibility of achieving consensus in an asynchronous network. That said, Paxos comes pretty close, and the way it does that is through those exponential back-offs that we have in the protocol.

Paxos Liveness:

- A Paxos run consists of one or more trials (rounds) run by different nodes.
- 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. So it will continue the work the previous proposer has started.
- 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 just doesn't finish.
    - Example: Dueling proposers (see figure to the right)
    - The situation can be made to be extremely unlikely with random back-offs.



V. Multi-Paxos and Applications

The preceding protocol solves the basic problem of consensus on a write-once register. Real consensus problems in distributed systems don't look like that, because useful data structures are rarely write-once. But the idea is that many useful data structures can be reduced to a set of write-once registers.

Replicating the WAL. The simplest to think about is the transactional log of an ACID database (the WAL). One way to think about the WAL is that it's a file whose value keeps getting updated or written to. The Paxos protocol we presented above is hardly amenable to that, as once you reached consensus on a value, there's no way to change it. But the other way to think about the WAL is that it's a **long vector of write-once registers**: each index in the log (record #1, #2, …) is written once and never updated – that's how a log operates. You start out with an empty log, i.e., no value is assigned to any of the indexes, then as you make progress on your transactions, you keep filling the values at the various indexes with records that the DB wants recorded in the log (e.g., begin, update, commit, prepare, … records). For each location, we can use the preceding Paxos protocol so the replicas agree on what value (record) is recorded at a particular index in the log. We'll have to amend the protocol a bit, for example

the Va, Na variables are now not just one, but a vector, one entry for each index in the log (Na^i, Va^i). Also, all the messages will need to refer to the "register" that they are writing, i.e., the specific entry in the log, i. This extension of Paxos is called *multi-Paxos*.

Now, suppose one of the replicas interacts with a client, and the client performs an update operation as part of a transaction. The replica becomes a Proposer, finds the next unused index in his log, call it *i*, and sends PROPOSE(N, i) to the other replicas. If that index is already "taken" by another record that the Proposer node didn't know about, then he will find out about that record when the other replicas reply to him with their N_a^i, V_a^i's. He will finish the protocol as before, not altering the value at index i in the log. In the process, he will amend his own knowledge about what indexes are filled in the log. Next, given that he didn't manage to finish the operation that his client asked him to make, he will again become a proposer, choose the next free index, j (could be i+1 or more, depending on how many records this node has missed), and run with that. He returns to his client only after he was able to pass through Paxos the value of his client's update at some index.

The preceding protocol has some inefficiencies. First, if clients are allowed to interact with different replicas of a shard to pass their operations, then proposers may conflict with each other frequently, causing a lot of useless work. Second, each time a replica becomes a proposer, he needs to complete the first phase of the protocol (PREPARE) in which he gets agreement on his sequence number N. It turns out that both issues can be addressed with a simple-looking (but actually major!) modification of the protocol.

Leader-Based Multi-Paxos. Once a replica completes one PROPOSE(N) phase (not tied to a specific index), it assumes the role of the "Leader." All the other replicas that know he's the Leader refuse client requests and forward the clients to the Leader. The Leader performs these updates by directly skipping to the ACCEPT(N, i) phase, reusing the proposal #N many times, but referring to different locations in the log every time. As long as the Leader keeps getting his proposals accepted by a majority, he can continue serving as a Leader (means a majority of the nodes agrees he's the Leader). If for a period of time he doesn't get any requests from his clients, he needs to send no-op messages to its replicas to renew his status as a Leader. If a particular period of time passes without the Leader getting acks from a majority of replicas, he stops behaving as a leader. Another node will take over by completing its own PROPOSE(N'>N) phase. However, his request to become a Leader will only be successful after a majority of the nodes agree that enough time has passed since they haven't talked with the former Leader to guarantee that the former Leader is no longer serving as a Leader. So the way to think about this is that the replicas must delay their responses to a PROPOSE message from a new node.

The preceding mechanism is called *Leader-based multi-Paxos* and it operates on the notion of ***leases***, a very important concept in DS. A lease gives a node permission to act in a certain role for a period of time, with the possibility of renewing the lease within that time frame. It's used throughout DSes as an optimization, just as we use it above. But a lease has very significant implications, because to function correctly, you have to make assumptions about maximum skew between the clocks at different nodes – please reflect on this! This is why leases must not be too long, or else the skew can add up. Typical leases are 30 seconds or so, during which time they need to be renewed or they are lost.

Other Applications:

- **Primary Election**: A common replication architecture is the Primary-Secondaries architecture, where one primary replica receives all read/writes to a particular object, assigns an order to the writes, and forwards them to the secondaries, which apply them in order. The condition is always that there's at most one replica behaving as primary for a given object at any given time. In HW2 you wrote your own protocol for how to ensure this condition – and you probably saw how hairy it can get even when the protocol itself was assisted by a magical viewserver that could never die. The way you would do primary election in the real world would be with a protocol like multi-Paxos (or some abstraction implemented on top of it), plus the notion of leases. The sequence of views that the system can go through is the vector of "write-once registers" in this case, and multi-paxos is used to assign to view i a primary value. Leases will be likely involved, as well.

- **Distributed Locks**: We talked about a protocol to do distributed locking in the Lamport Clocks lecture. That protocol doesn't have good fault tolerance (see that lecture). In the real world, the way distributed locks work is that with a distributed locking *service*. It's the same as having a distributed lock *server*, but with fault tolerance achieved through Paxos-based replication. Each lock has an associated vector of write-once registers of who holds the lock at a particular iteration… Etc.

Google's Chubby lock service, which we discuss next time, is an example of such a lock service that uses Paxos to ensure fault tolerance of the locks. Interestingly, one use case for the Chubby service that we'll talk about is *primary election*: the primary takes a lock from the lock service and retains it for the period of the lease (and tries to refresh the lease by periodically re-taking the lock). So yes, primary election usually relies on a consensus protocol like Paxos, but in practice it's often not implemented directly on Paxos but rather layered on top of some other Paxos-based service. The same for other types of problems, including replication of the WAL.

VI. RAFT

RAFT is a leader-based consensus protocol that streamlines many of the preceding concepts (multi-paxos, leader election, log replication). It thus provides a higher-level of abstraction with one integrated protocol. It is generally deemed as simpler to understand and more actionable than the preceding protocols. A lot of companies have started to adopt it. We won't cover it in class, but please watch the presentation from USENIX ATC 2014: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>. Feel free to also read the corresponding technical paper, but only information from the presentation is relevant for the final quiz.

Key Papers

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

Acknowledgements

The preceding notes contain portions adapted from the NYU distributed systems course, Jinyang Li's edition: http://news.cs.nyu.edu/~jinyang/fa10/notes/ds-paxos.ppt.