CONSENSUS PROTOCOLS FOR FAULT TOLERANT DATABASES

Last time we talked about *sharding* – a key mechanism for scaling distributed systems – and the semantic challenges raised by that mechanism. The main challenge there is how to execute groups of related operations atomically (either all or none) when different shard servers are responsible for executing subsets of these related operations. We presented the two-phase commit (2PC) protocol for coordinating nodes of a sharded database to support atomic database transactions.

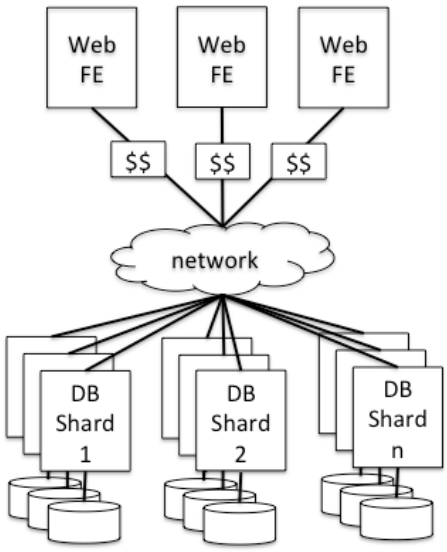
Today we'll focus on *replication* – a key mechanism for fault tolerance in distributed systems – and the semantic challenges raised by that mechanism. We will show that, perhaps unintuitively, the challenges are very different from those raised by sharding and require a very different type of protocol to address them. I.e., we can't just change 2PC to deal with semantic issues raised by replication (despite years' worth of efforts to do just that in the '80s). We will look at one particular protocol for how to address semantic issues raised by replication, called *the RAFT consensus protocol*. Next time we'll put the two together by looking at the design of Spanner, Google's fault tolerant and scalable ACID storage 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.

IV. THE RAFT CONSENSUS PROTOCOL

RAFT is a leader-based protocol, where one replica is elected as leader (coordinator from our 2PC-based "proposed protocol"). The other replicas are called followers. The protocol includes a leader election protocol plus a log replication protocol. The protocol's functioning is described in this presentation <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>.