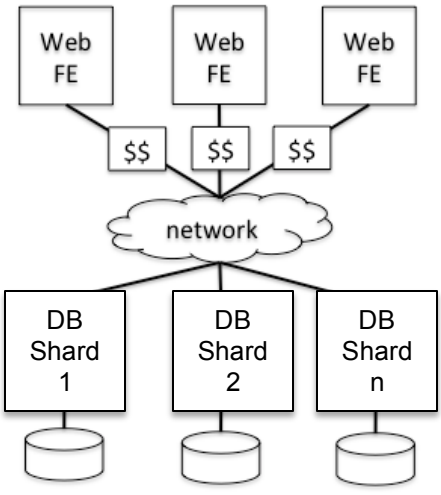
ATOMIC COMMITMENT
Or: How to Implement Distributed Transactions in Sharded Databases

We talked about transactions and how to implement them in a single-node database. We'll now start looking into how to implement transactions in distributed databases. It will be our excuse to discuss atomic commitment, and subsequently, consensus protocols.

We distribute things, including databases, for two reasons: for scalability and for fault tolerance. For scalability, we shard the workload; in the case of databases, we often shard the tables by row. For fault tolerance, we replicate the workload; in the case of databases, we replicate each shard. As we've said before, both of these mechanisms raise semantic challenges. The core protocol used to address the semantic challenges raised by sharding a database is two-phase commit (2PC), an atomic commitment protocol. One protocol used to address some of the semantic challenges raised by replicating the shards of a database is the Paxos protocol, a consensus protocol. Over the next few lectures, we'll discuss these two protocols from the perspective of their application to distributed transactions. In this particular lecture, we focus on sharding and the challenges of implementing transactions on sharded databases, plus how to layer the 2PC protocol on top of non-sharded transactional databases. So, for this lecture, ignore replication!

I. SEMANTIC CHALLENGES

Let's go back to the Web service example we presented in one of our original lectures. Let's revisit the last architecture we discussed, but ignore the replication aspect of it. For concreteness, let's specify what the application (implemented by the FE) is in this case: say it is a banking application, where user account information is stored in the DB and users can perform money transfer transactions to move data from one account to another as long as they have sufficient funds in the source account.



Architecture: FE and DB are both sharded. FEs accept requests from end-users' browsers and process them concurrently (i.e., two requests from the same or from different users can execute in parallel on two different FEs). The data stored in the DB is *sharded*, say by user ID. All data of the first third of the users in ID space is stored in Shard 1; all data of the next third of the users in ID space is stored on Shard 2; and all data of the last third of the users in ID space is stored on Shard n.

First question: How does this architecture increase the capacity of your system (i.e., the maximum load the system can handle)? Answer: As long as most of the workload involves accessing accounts from one of the shards (e.g., users check out mostly their accounts), the overall capacity should increase, because you now have the various DB shards handling data requests in parallel from multiple different FEs.

Second question: Suppose each individual shard implements an ACID engine. What are the challenges of implementing ACID transactions across the entire service as a whole?

Answer: If transactions never span multiple shards (i.e., access rows across multiple shards), then no challenges. Otherwise, individual shards participating on any transaction need to agree on (1) whether or not to commit a transaction and (2) when to release the locks. Let's take an example. Suppose a user wants to transfer some money from his account to another user's account. This bank transaction involves two operations on two different parts of the database – deducting the money from the source account and adding it to the destination account. Sometimes, these two accounts will be stored on different machines. From a semantic perspective, it's important that both operations either succeed or fail, otherwise you can either "lose" money (if the operation completes on the source but not on the destination) or "create" money out of thin air (if the operation fails on the source but completes on the destination). Unfortunately, the two machines are independent and hence they can fail or decide to unilaterally abort *independently*, so what we would like is a coordination protocol between the two DB shard servers that lets us ensure that the two operations can either both succeed, or if one fails, the other one is not applied, either. **That is what atomic commitment protocols give you, and the best known such protocol is Two-Phase Commit (2PC).**

II. TWO-PHASE COMMIT (2PC)

What the single-node transactional support gives us is a way to implement the various operations included in a transaction, in an atomic and isolated way, against one of the nodes. In the sharded case, a transaction will consist of several operations, subgroups of which will be executed against different servers. We would like to preserve the same atomicity, durability, and isolation semantics as we did in the single-node case, but in a sharded case now. What we'll do is to execute each portion of the transaction that is relevant for each shard using the techniques from the previous lecture, and then we'll devise a protocol for the various shards involved in a transaction to decide whether they should all commit their portions of the transaction or not. The protocol is called **two-phase commit**, and we'll first describe it in the context of the example in Section I. After that we'll generalize.

Simplified 2PC based on Section I Example

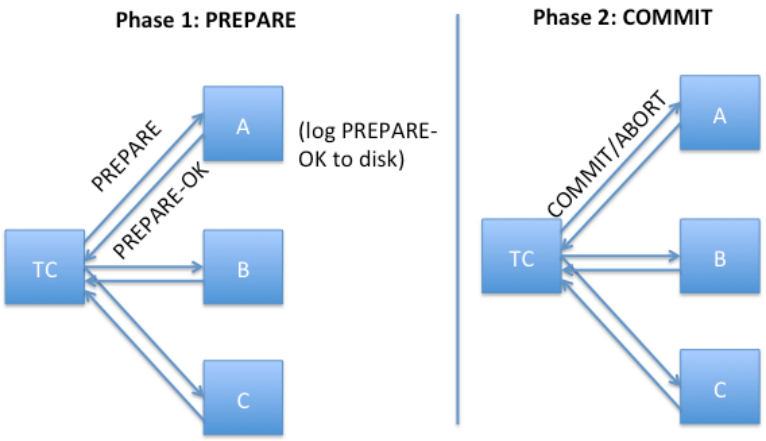
Suppose you have a simple transaction, which consists of two operations only: deduct money from the source account (op1) and add it to a destination account (op2). And suppose the source and destination accounts are stored on different servers (due to DB sharding), S1 and S2, respectively. The operation relevant to S1 will be op1; the operation relevant to S2 will be op2. We would like either both or neither of these operations to be executed on the two servers. We'll do several things:

1. To **begin** a distributed transaction, the client (in this case one of the FEs, on behalf of the end user) initiates transactions on each separate shard server.
2. **As part of the distributed transaction**, the client will send the operation to the corresponding shard server. Op1 goes to S1 and op2 goes to S2. Each shard will perform these operations as we described in the previous section: S1 will lock the row for the source account, will log sufficient information about op1 to its write-ahead log to be able to undo op1 if necessary, and finally will perform the operation on the database; S2 will do similarly for op2. The two servers maintain their respective locks until the end of commit (Step 3.4. from below).

3. When it's time to **commit** the transaction, we will proceed as follows:
   3.1. The client (or rather, the DB library on behalf of the client code) sends a **PREPARE** message to each shard server, specifying the transaction ID.
   3.2. Upon receipt of a **PREPARE** message for a transaction txID, a shard server, Si, will: (1) determine whether it can commit the transaction, (2) write whether it can to its write-ahead log (this will be a PREPARE-OK or PREPARE-FAIL entry in the log); and (3) reply with **PREPARE-OK** or **PREPARE-FAIL**, respectively, to the client. For example, if the server has failed some time before receiving PREPARE from the client, or if it had to abort the transaction to resolve some deadlock, or for some other reason, like finding that one of the accounts has been disabled in the meantime, then the server will send a **PREPARE-FAIL** response back to the client and will proceed to undo the transaction based on its write-ahead log and release all of its locks. If, however, the server found nothing wrong with the transaction at the time it received PREPARE from the client, then it will send a **PREPARE-OK** in this step to the client. In that case, the node must wait for a next message from the client before it releases its locks and marks the transaction as committed in the write-ahead log. If after a restart, a node finds a PREPARE-OK entry in its log for a transaction, it cannot unilaterally decide to abort.
   3.3. On the client, upon receiving PREPARE-OK/PREPARE-FAIL responses from the shard servers:
      3.3.1. If the client received **PREPARE-OKs from both servers**, then the client sends a **COMMIT** message to both servers S1 and S2. It then waits again for the acknowledgement.
      3.3.2. If the client receives **one PREPARE-FAIL response**, then the client will send an **ABORT** message to both servers S1 and S2.
   3.4. Upon receipt of a **COMMIT** or **ABORT** message from the client, a shard server, Si, will: (1) enter commit/abort in its write-ahead log, (2) if it's ABORT then it goes on and reverts the effects of the transaction on the database, and (3) release the locks it was holding for the transaction.
4. To **abort** a distributed transaction, the client sends the ABORT message to S1 and S2.

More General Version of the Protocol

The protocol we presented above is driven by our example and is a simplification of the actual 2PC protocol. More generally, two 2PC protocol is performed across two or more nodes (called *participants)*, and in the picture below, which illustrates the more general version, it is performed across four participants. Also, in more general settings, it is a bad idea to have the client – an external entity w.r.t. the DB service – coordinate the transaction. Instead, it helps if one has a designated server, called a *transaction coordinator*, send the PREPARE



and COMMIT messages. The coordinator can then record the various phases at the protocol into its own log to help with recovery from various conditions (we'll discuss recovery next). The figure to the side shows the transaction coordinator (TC) and the two phases of the 2PC protocol, each corresponding to a message exchange. The four total messages (1. PREPARE, 2. PREPARE-OK/FAIL, 3. COMMIT/ABORT, and 4. COMMIT-OK/ABORT-OK) correspond to the four steps (3.1-3.4) in the example before.

Handling Timeouts/Failures

The preceding protocol description deals with "happy cases," but doesn't specify what happens on various failures. Let's study those situations next. Focus on the description of **commit()**, with the four steps denoted there 3.1-3.4. These four steps are denoted in what follows simply as 1-4.

Timeouts:

- The waits are before steps 2, 3, and 4.
- 2: the participant times out before it has responded PREPARE-OK/FAIL. So it's safe for it to abort on timeout.
- 3: TC times out waiting for PREPARE-OK or PREPARE-FAIL from participants; so, safe to abort on timeout
- 4: A participant p times out before it received the outcome from TC (COMMIT/ABORT). If p voted PREPARE-FAIL at the previous step, then p can abort. If he voted PREPARE-OK, then p finds itself in an **uncertainty** period. He can't abort or commit. He needs to find what the decision actually was. So, p needs a **termination protocol**:
    a) "wait until communication with TC is re-established." Safe but downside is the participant may be blocked unnecessarily, since it can learn the decision from any other participant that has decided.
    b) "cooperative" -- participants know about each other, and p pings q for outcome. (if q is not in uncertainty period and has not decided, q can abort as the outcome!)

Recovery:

- if participant is not in uncertainty period, on recovery, can decide what to do. (unilaterally abort if no decision, otherwise do what decision is.)
- if participant is in uncertainty period, it cannot decide on its own, must run **termination protocol** (as above).

What state must be stored reliably on disk (i.e., when does the log need to be sync'ed)?

- Coordinator: Records "PREPARE" record in its log, but it doesn't need to flush, and it doesn't matter if it writes this entry before or after it sends PREPARE to participants. The reason it should write PREPARE is so the coordinator knows upon recovery that it has started the transaction so it needs to finalize it. Why is it OK to not flush to disk this entry? Because if upon recovery this entry is absent, this means that the COMMIT entry is also absent (see below), hence the coordinator knows it could not have sent a decision to any of the participants. So it can unilaterally decide to abort this transaction.
- Participant: If votes PREPARE-OK, must write a "prepare-ok" record in log and sync to disk before sending PREPARE-OK vote to coordinator. If votes PREPARE-FAIL, must write a

"prepare-fail" record in log, but can be either before or after sending PREPARE-FAIL vote and doesn't need to sync. (Why? Because if no "prepare-ok" or "prepare-fail" record, unilaterally abort.)

- Coordinator: Before the coordinator sends COMMIT, it must record commit record in its log and sync to disk. If the coordinator sends ABORT, it writes an abort record, but doesn't need to sync and can write the record before or after sending the messages. (same argument as above)
- Participant: After receiving COMMIT or ABORT, records the decision in the log, but doesn't need to synch and can process transaction before or after recording the decision.
- At what point has the commit happened? After the commit record hits the log of the coordinator on disk. That is when the coordinator can also rely to the client with the "committed" outcome (see transactional API).

When is it safe to prune logs?

- site cannot delete log records of a transaction until the commit or abort has been processed.
- at least one site must not delete transaction's records from log until that site has received messages from everybody saying that their commit/aborts have been processed. This is so that recovery is always possible.

III. PROPERTIES OF 2PC

Performance

- It's expensive! You're holding locks while executing a distributed protocol! If you're distributed over geographies, that's really expensive!!
- Time complexity:
  - 2PC requires three message delivery latencies: PREPARE → YES/NO → ABORT/COMMIT
  - on failure, additional rounds may be necessary to recover
- Message complexity
  - common case for n participants plus 1 coordinator: 3n messages are sent

Fault Tolerance

- **Not great!**
- Availability problems: a process can block indefinitely in its uncertainty period (after it responds PREPARE-OK to TC's proposal but before it receives decision from TC), until the failure is resolved.
  - Biggest vulnerability: until coordinator hears "PREPARE-OK" from everybody, everybody is uncertain. If coordinator goes down after everybody has voted PREPARE-OK, but before coordinator has sent a decision to anyone, everybody blocks. Thus, a **single-site failure can cause 2PC to block indefinitely**!
  - And it blocks while each shard is **holding locks**, preventing other transactions that don't even interact directly with the failed shard server from making any progress.

It is said that **2PC is a "blocking" protocol**, because even with one node failure, it may not be able to make progress until the failure resolves itself. By contrast, a "live" (a.k.a., fault tolerant) protocol would make progress, at least despite individual-node failures. The way 2PC is used in practice is to layer it **on top of replication**, such that individual shards become replicated and fault-tolerant services that appear to **never fail**. So, we eliminate failures from 2PC and it becomes a useful (though still expensive) protocol. See the architecture figure in the beginning of the notes.

Next lecture we'll start looking at protocols in support of replication, starting with protocols for consensus.

Key Papers

[Lampson-Sturgis-1979] Butler Lampson and Howard Sturgis. *Crash Recovery in a Distributed Data Storage System*. In *Distributed Systems— Architecture and Implementation*, 1979.

Acknowledgements

The preceding notes contain portions adapted from the University of Washington distributed systems course, Steve Gribble's edition:
https://courses.cs.washington.edu/courses/csep552/13sp/lectures/4/2pc.pdf.