490H: Two phase commit (note this is different from two phase locking)

Wrap up from last time:

With two phase locking, where do we do the commit? At the client, or the server? Either! If at the client, that means that any client can be blocked from making progress because some other node in the system has crashed – if it has a read or write lock on some file, and then crashes, other nodes won't be able to know if the change has committed until the node recovers. If we send all changes to the server to perform the commit, then clients can continue if another client fails (provided the server can revoke the failed client's locks, by telling it that the transaction crashed), but the clients will still be blocked if the server fails.

With optimistic concurrency control, where do we need to do the commit? The server – some single location has to decide if there is a conflict between the independent activities of the various clients. So the easy way to implement this would be to send all modified files back to the server, and then ask it to commit once it has stored those changes. But if we are willing to have the clients block,

So far: we use caches and cache coherence to be able to do operations more quickly, and transactions to ensure that the persistent state is updated at the server in a consistent way, despite client failures.

What if we need to update state at two servers? We still want the state to be updated consistently and atomically, despite client and server failures.

[We might need multiple servers for various reasons: eg., because the state was too large to store on one server, or there was too much traffic for one server to handle, etc. Another reason is that we might want to update state in multiple organizations (as in a bank transfer). ]

How can we update state in two places in an atomic, consistent way?

Recall the two generals problem from the first class: we showed that you can't coordinate simultaneous action on two nodes, in the presence of unreliable messages, even if no messages get lost.

So is it hopeless?

If we can't coordinate simultaneous action, what can we have?

Eventual agreement, provided nodes eventually recover.

That's two phase commit.

Coordinator:

Log transaction start
Send vote-req to participants
Get replies
If all yes, log commit
If not, log abort
notify participants

at participant:

wait for vote-req
determine if transaction can commit locally (no deadlock, enough space, etc.)
log result
send result to coordinator
if result is ok to commit, wait for commit/abort from coordinator
log what coordinator says

Walk through algorithm: what happens if failure at each step?

What if we did things slightly differently? E.g., do we need the message log at the participants? What if we reply then log?

What properties does 2pc have?
Safety:
1. All hosts that decide reach the same decision.
2. No commit unless everyone says "yes".

Liveness:
3. No failures and all "yes", then commit.
4. If failures, then repair, wait long enough, then some decision.

(marriage anecdote)

How well does 2PC work in practice? Well, not so well, and for reasons that are clear from the optimistic concurrency example we did earlier.

That is, it is a blocking protocol – if the coordinator fails, everyone needs to wait for the coordinator to recover to discover whether the commit occurred.

In essence, we've turned the problem of updating state in multiple locations atomically, into a single commit at the coordinator. That means we depend on the coordinator, and if it fails, we're stuck until it recovers and can tell us what happened.

Applications:

Example: send mail
Two ways to build this: in one, write email to server, server provides to clients
Alternate: clients communicate directly with each other using 2pc to build a reliable RPC

walk through send mail to many users example
what if one user doesn't exist?
but mail has already been delivered to some other users
how to un-do?
what if concurrently one reader reads his/her mail?
how does user not see tentative new mail?
does reading user block? where?
read_mail also deletes it
what if new mail arrives just before delete?
will it be deleted but not returned?
why not? what lock protects? where is the lock acquired? released?
what if a user is on the list twice?
locks are held until end of top-level xaction
deadlock?

Next question: can we design a non-blocking commit protocol? That is, even though nodes can fail and restart, and messages can be delayed and lost, we can still make progress when a node fails without waiting for it to restart?

That's Paxos, topic for next week.

We're going to replace the single server of the project, with a set of nodes doing the same function as the server. Each server replica will be identical, but together we need to have them work in concert to decide on a commit, so that we can figure out whether the commit happened, even if any node fails (or is so slow it appears to have failed).

Should seem hard! But this is a key building block in almost all highly available distributed systems today.