Distributed Transactions

Recap:

Transaction definition: group of operations with four properties

Atomic – all or nothing
Consistent – equal to some sequential order
Isolation – no data races
Durable – once done, stays done

A key idea is that it is ok to start over – correctness is more important than progress.

Atomicity and durability by converting group of operations into a single update to disk – logging or an atomic switch to a shadow copy

Isolation and consistency: two phase locking or optimistic concurrency control

Example:

Tx1:
read x
Compute
Write y
Commit

Tx2: (during the compute phase of tx1)
Read y
Write x
Commit

These two transactions conflict – tx2 has to come either completely before or completely after tx1.

Two strategies: We can hold up the second transaction until the first commits (two phase locking), or we can go ahead and commit the second, and when the first tries to commit, we can tell it "sorry!" and have it abort and retry (optimistic concurrency control).

Approach 1: two phase locking. Every read/write "locks" that location for the duration of the transaction. In other words, the protocol is "blocking" – we need to hold up serving reads and writes until the transaction completes, so that we can be sure to achieve serializability.

Why do we have to hold the lock for the duration of the transaction?

Otherwise, could see the output of the transaction even though a later failure caused it to abort. It is only safe to read the results of a transaction after the commit occurs.

With locking, you might need to worry about deadlock – e.g., in the case above, we can deadlock if we acquire the locks as we need them. Here's the great thing about transactions: Can always break a deadlock by aborting one of the waiting transactions. If we revert back to the start, we can then retry.

[Note! This means that a transaction can abort, due to no fault of its own! So your implementation needs to handle that case.]

Similarly, if you are implementing a file system, and somewhere inside a file system operation, you find that there's no free block so you can't complete the operation (e.g., for a file move)? Transactions give you a structured way to handle exceptions, in the presence of durable storage.

<anecdote about friend at the terminal> Solution? Write new file, then delete old file, so you can back out if there's a problem. Never overwrite important data.

[If we batch commits, then we might want to allow the second transaction to start without waiting for the commit. To be precise, its ok to allow one transaction to read the results of another, if we ensure that they are chained – that it will abort if the first one aborts.]

Approach 2: Optimistic concurrency control (spiffier, and how we suggest you do the assignment)

[Synopsis of approach 2: With two phase locking, locks are held during the entire commit procedure. Logically that's ok, but if those locks have contention, things can be slow. So maybe do something optimistic: a sequence of versions of each bit of state, where transactions commit against a logically consistent set of versions (at a logical timestamp).]

Analogy: source code version control. Take a snapshot of the state of the system, modify what you need, then check it back in – if in the meantime someone else has also modified it? Then restart from the new version. Maybe get unlucky again! Just keep trying.

In approach 1, we needed a way to revert to the beginning of the transaction to deal with deadlock. This means we have to keep track of multiple versions of each file, so that we can revert correctly. Given that, maybe we can simplify things a bit? Suppose we let other nodes proceed with their reads and writes, without any locking. Then we can check at the point we do the commit, whether there is a conflict, and if so, roll that particular transaction back.

What do I mean by "if there is a conflict"? To be serializable, we need each transaction to complete in a single order everywhere.

One can think of transactions as committing in that order T1, T2 … Ti, Ti+1, … and so forth.

We can commit Ti, provided that all of its reads and all of its writes are consistent with the state of the storage system at that time. So if we read a file, we need it to be the version of the file that is live after transaction Ti-1.

Why might this not be true? Well, if we don't have locks, we might have allowed some other transaction to read a file we modified, or to write a file we had read. If that transaction finishes before we do, it means one or the other of us needs to abort. You can prove that you make progress by showing that one or the other will succeed – if someone invalidates your reads by committing a write before you are able to commit, then it means that some transaction committed, and there's progress. Likewise, if you aren't able to commit a write because you depend on someone else's writes, that means they'll be able to commit, even if you needed to abort.

The simple way to think about this is that you pick Ti at the time of the commit, after all other commits so far. But we do have a bit more flexibility: we are safe picking an earlier virtual time, Tj < Ti, if the reads are consistent with the state of the storage system at time Tj, and the writes hadn't been read by any transaction after Tj.

To make this work, we need to keep track of the versions of each file that are read or written as part of any transaction.

So for example, computing the sum of all bank balances becomes easy – compute on the old version of the bank balances, allowing later updates to proceed. We can then garbage collect the old bank balances when the sum finishes.

Other transactions can start and even finish, as long as they don't depend on the output of the first transaction; that is, the sum can commit or abort independently of later transactions, because it doesn't modify any of the account balances. (This is still linearizable – the transactions appear to be done, in an order consistent with the range of times that each of the transactions executed. If we allow time travel transactions – transactions to be done on the database as if they were done in the distant past -- then its only serializable.

We are always safe to abort transactions whenever linearizability would be broken. That is, we can allow all transactions to proceed, as long as we can detect if there is a problem, and abort any transactions as necessary.

What about deadlock? In optimistic concurrency control, we never have deadlock! We let everyone proceed, and abort any transaction that would violate consistency. Either way, we need to be able to abort a transaction.

Here's the twist: recall that we can't tell if a client is slow or if it has crashed. So if we're going to make cache coherence work with client failures, we need to be able to revoke a cached copy of a file. Otherwise, we'll need to stop everything until the client recovers! But maybe the client was just slow, and we revoked the copy incorrectly.

So we need a mechanism to be able to detect whether a commit can be allowed to go forward at the server – did we revoke the client's data correctly (it had crashed) or incorrectly (it hadn't). With optimistic concurrency control, its automatic – we simply don't care: we abort any transaction that can't commit because it is using stale or invalid data.

--

Distributed transactions and two phase commit

So far, we've been updating state at a server. 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.

But what if we need to update state at two servers? An example: data is stored in shards across a number of servers, but we still want serializability across the entire system. We want the state to be updated consistently, despite client and server failures.

Now recall 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 how are we to update state in two places in a consistent way?

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

Eventual agreement, provided nodes eventually recover.

We could just have one side dictate the result – but then, if the other side can't complete the transaction (e.g., it runs into a deadlock), problem! Hence, two phase commit – first, check that transaction can commit everywhere, then one node (the coordinator dictates the result).

Coordinator:

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? 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 of two phase commit

How would you build a reliable, p2p send mail application?

Two ways to build this: in one, write email to server, server provides to clients
Alternate: clients communicate directly with each other using 2pc

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.