550: Transactions

Main points

Transactions on one node
Transactions across multiple nodes

Atomicity abstraction: ACID

Atomic – all or nothing
Consistent – equal to some sequential order
Isolation – no data races; transactions need locks (not discussed in paper)
Durable – once done, stays done

What do we mean by consistent? We have all the options from last week, but now for groups of operations, rather than individual ones. That is, transactions could appear eventually consistent, causally consistent, sequentially consistent (in database terms, serializable), or linearizable (a single copy, consistent with the operations in real time).

Since the transaction commits at a single point in time, when the commit hits the physical disk, and the transaction groups operations that are logically part of one unit, it makes sense to treat all of the operations inside the transaction as "happening" as if they occurred at the time of the commit.

Example:
Move a file, while doing grep or a backup. If the grep starts after the file move finishes, then should appear after after the move. Otherwise, grep should see the file in the original location, but not in both or neither.

[Or say, backup: how do you know your backup is getting a consistent view, if the file system is being modified while the backup is going on? With shadow files, easy!]

Approach 1: two phase locking. Every read/write "locks" that location for the duration of the transaction. If both the file move, and the grep are done as part of separate transactions, then this will prevent grep from reading an inconsistent value during the move.

In terms of the assignment, this is equivalent to saying that once you grant exclusive access to a file to a client, you need to let the client keep it until it completes its transaction (invalidation must be delayed). Similarly, if a file is being read by a transaction, so it is in the cache read-only, you can't invalidate it for a competing write.

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.

OK, do we need to worry about deadlock? [Do we need to worry about deadlock in assignment 2? No – while we might need to queue read or write requests, as long as there are no failures, each request can be satisfied.] Yes!

Example:

Tx1: move file from A to B
Tx2: move file from B to A

Tx1 could grab exclusive access to A, and then try to get B; meanwhile, Tx2 could grab exclusive access to B, then try to get A. Ooops.

Here's the great thing about transactions: Can get deadlock, 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.]

[another example: compute the sum of bank balances in a bank. With two phase locking, it will stop all transactions on any bank account until the transaction completes! Ugh!]

Approach 2: Optimistic concurrency control (a bit spiffier, and how we suggest you do it in assignment 3)

[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 you'd rather 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).]

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, it 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. It might be sequentially consistent (in database lingo, serializable) if we allowed time travel transactions – transactions to be done on the database as if they were done in the distant past. Of course, they are only serializable if no other transaction committed at a later virtual time, having depended on the earlier transaction having not occurred.)

We are 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.

For 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. We can hold up the second transaction until the first commits (akin to 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.

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 we need to be able to revoke a cached copy of a file, that is read only or writeable, in the client's cache. If it has crashed, we're safe, but if it is slow, it means that it will eventually get around to trying to finish its transaction. That transaction probably needs to be aborted.

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 whatever 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? We still 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.

That's two phase commit. (marriage anecdote)

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?

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 recovery to discover whether the commit occurred. Of course, we can't tell if the coordinator has failed, or it is just being slow.