Distributed Transactions FAQ

Q: How does this material fit into 6.824?

A: When data is distributed over many computers, it's common for a single operation to need to read and/or modify data on multiple computers. How such multi-step operations interact with concurrent operations on the same data, and what happens if a crash occurs in the middle of such an operation, are usually critical questions for the system's robustness and ease of programming. The gold standard for good behavior is transactions, often implemented with two-phase commit, two-phase locking, and logging. Today's reading from the 6.033 textbook explains those ideas. Later we'll look at systems that provide similarly strong semantics, as well as systems that relax consistency in search of higher performance.

Q: Why is it so important for transactions to be atomic?

A: What "transaction" means is that the steps inside the transaction occur atomically with respect to failures and other transactions. Atomic here means "all or none". Transactions are a feature provided by some storage systems to make programming easier. An example of the kind of situation where transactions are helpful is bank transfers. If the bank wants to transfer $100 from Alice's account to Bob's account, it would be very awkward if a crash midway through this left Alice debited by $100 but Bob *not* credited by $100. So (if your storage system supports transactions) the programmer can write something like

```
BEGIN TRANSACTION
  decrease Alice's balance by 100;
  increase Bob's balance by 100;
END TRANSACTION
```

and the transaction system will make sure the transaction is atomic; either both happen, or neither, even if there's a failure somewhere.

Q: Could one use Raft instead of two-phase commit?

A: Two-phase commit and Raft solve different problems.

Two-phase commit causes different computers to do *different* things (e.g. Alice's bank debits Alice, Bob's bank credits Bob), and causes them *all* to do their thing, or none of them. Two-phase commit systems are typically not available (cannot make progress) in the face of failures, since they have to wait for all participating computers to perform their part of the transaction.

Raft causes a majority of the peers to all do the *same* thing (so they remain replicas). It's OK for Raft to wait only for a majority, since the peers are replicas, and therefor we can make the system available in the face of failures.

Q: In two-phase commit, why would a worker send an abort message, rather than a PREPARED message?

A: The reason we care most about is if the participant crashed and rebooted after it did some of its work for the transaction but before it received the prepare message; during the crash it will have lost the record of tentative updates it made and locks it acquired, so it cannot complete the transaction. Another possibility (depending on how the DB works) is if the worker detected a violated constraint on the data (e.g. the transaction tried to write a record with a duplicate key in a table that requires unique keys). Another possibility is that the worker is involved in a deadlock, and must abort to break the deadlock.

Q: Can two-phase locking generate deadlock?

A: Yes. If two transactions both use records R1 and R2, but in opposite orders, they will each acquire one of the locks, and then deadlock trying to get the other lock. Databases are able to detect these deadlocks and break them. A database can detect deadlock by timing out lock acquisition, or by finding cycles in the waits-for graph among transactions. Deadlocks can be broken by aborting one of the participating transactions.

Q: Why does it matter whether locks are held until after a transaction commits or aborts?

A: If transactions release locks before they commit, it can be hard to avoid certain non-serializable executions due to aborts or crashes. In this example, suppose T1 releases the lock on x after it updates x, but before it commits:

```
  T1:           T2:
  x = x + 1
                y = x
                commit

  commit
```

It can't be legal for y to end up greater than x. Yet if T1 releases its lock on x, then T2 acquires the lock, writes y, and commits, but then T1 aborts or the system crashes and cannot complete T1, we will end up with y greater than x.

It's to avoid having to cope with the above that people use the "strong strict" variant of 2PL, which only releases locks after a commit or abort.

Q: What is the point of the two-phase locking rule that says a transaction isn't allowed to acquire any locks after the first time that it releases a lock?

A: The previous question/answer outlines one answer.

Another answer is that, even without failure or abort, acquiring after releasing can lead to non-serializable executions.

```
  T1:         T2:
  x = x + 1
              z = x + y
  y = y + 1
```

Suppose x and y start out as zero, and both transactions execute, and successfully commit. The only final values of z that are allowed by serializability are zero and 2 (corresponding to the orders T2;T1 and T1;T2). But if T1 releases its lock on x before acquiring the lock on y and modifying y, T2 could completely execute and commit while T1 is between its two statements, giving z a value of 1, which is not legal. If T1 keeps its lock on x while using y, as two-phase locking demands, this problem is avoided.

Q: Does two-phase commit solve the dilemma of the two generals described in the reading's Section 9.6.4?

A: If there are no failures, and no lost messages, and all messages are delivered quickly, two-phase commit can solve the dilemma. If the Transaction Coordinator (TC) says "commit", both generals attack at the appointed time; if the TC says "abort", neither attacks.

In the real world, messages can be lost and delayed, and the TC could crash and not restart for a while. Then we could be in a situation where general G1 heard "commit" from the TC, and general G2 heard nothing. They might be in this state when the time appointed for attack arrives. What should G1 and G2 do at this point? I can't think of a set of rules for the generals to follow that leads to an acceptable outcome across a range of situations.

This set of rules doesn't work, since it leads to only one general attacking:

  * if you heard "commit" from the TC, do attack.
  * if you heard "abort" from the TC, don't attack.
  * if you heard nothing from the TC, don't attack.

We can't have this either, since then if G1 heard "abort" and G2 heard nothing, we'd again have only one general attacking:

  * if you heard "commit" from the TC, do attack.
  * if you heard "abort" from the TC, don't attack.
  * if you heard nothing from the TC, do attack. [note the "do" here]

This is safe, but leads to the generals never attacking no matter what:

  * if you heard "commit" from the TC, don't attack.
  * if you heard "abort" from the TC, don't attack.
  * if you heard nothing from the TC, don't attack.

The real difficulty in the dilemma is that there's a hard deadline at which both generals (or neither) must simultaneously attack. If there is no deadline, and it's OK for the participants (workers) to commit at different times, then two-phase commit is useful.

Q: Are the locks exclusive, or can they allow multiple readers to have simultaneous access?

A: By default, "lock" in 6.824 refers to an exclusive lock. But there are databases that can grant locking access to a record to either multiple readers, or a single writer. Some care has to be taken when a transaction reads a record and then writes it, since the lock will initially be a read lock and then must be upgraded to a write lock. There's also increased opportunity for deadlock in some situations; if two transactions simultaneously want to increment the same record, they might deadlock when upgrading a read lock to a write lock on that record, whereas if locks are always exclusive, they won't deadlock.

Q: How should one decide between pessimistic and optimistic concurrency control?

A: If your transactions conflict a lot (use the same records, and one or more transactions writes), then locking is better. Locking causes transactions to wait, whereas when there are conflicts, most OCC systems abort one of the transactions; aborts (really the consequent retries) are expensive.

If your transactions rarely conflict, then OCC is preferable to locking. OCC doesn't spend CPU time acquiring/releasing locks and (if conflicts are rare) OCC rarely aborts. The "validation" phase of OCC systems often uses locks, but they are usually held for shorter periods of time than the locks in pessimistic designs.

Q: What should two-phase commit workers do if the transaction coordinator crashes?

A: If a worker has told the coordinator that it is ready to commit, then the worker cannot later change its mind. The reason is that the

coordinator may (before it crashed) have told other workers to commit. So the worker has to wait (with locks held) for the coordinator to reboot and re-send its decision.

Waiting indefinitely with locks held is a real problem, since the locks can force a growing set of other transactions to block as well. So people tend to avoid two-phase commit, or they try to make coodinators reliable. For example, Google's Spanner replicates coordinators (and all other servers) using Paxos.

Q: Why don't people use three-phase commit, which allows workers to commit or abort even if the coordinator crashes?

A: Three-phase commit only works if the network is reliable, or if workers can reliably distinguish between the coordinator being dead and the network not delivering packets. For example, three-phase commit won't work correctly if there's a network partition. In most practical networks, partition is possible.

Q: Can there be more than one transaction active? How do participants know which transaction a message refers to?

A: There can be many concurrent transactions, managed by many TCs. A TC assigns a unique transaction ID (TID) to each transaction. Every message includes the TID of the relevant transaction. TCs and participants tag entries in their tables with the TID, so that (for example) when a COMMIT message arrives at a participant, it knows what tentative records to make permanent, and what locks to release.

Q: How does a two-phase commit system undo modifications if a transaction has to abort?

A: Each participant performs modifications to temporary copies of the records. If the participant answers "yes" to the TC's prepare message, the participant must first save the temporary record values to its log on disk, so it can find them if it crashes and restarts. If the TC decides to commit, the participant must copy the temporary values to the real database records; if the TC decides to abort, the participant must discard the temporary records.

Q: How does serializability relate to linearizability?

A: They are similar notions, arising from different communities. Both require the final outcome to be the same as some serial execution. Serializability usually refers to entire transactions, each involving multiple operations. Linearizability often refers to simple reads and writes. It's also the case that linearizability requires that the equivalent serial execution match the real time order of the actual execution, while serializability usually does not.

Q: Why do logs appear so often in the designs we look at?

A: One reason is that a log captures the serial order that the system has chosen for transactions, so that e.g. all replicas perform the transactions in the same order, or a server considers transactions in the same order after a crash+reboot as it did before the crash.

Another reason is that a log is an efficient way to write data to hard disk or SSD, since both media are much faster at sequential writes (i.e. appends to the log) than at random writes.

A third reason is that a log is a convenient way for crash-recovery software to see how far the system got before it crashed, and whether the last transactions have a complete record in the log and thus can safely be replayed. That is, a log is a convenient way to implement crash-recoverable atomic transactions, via write-ahead logging.

Q: Are there structures other than logs that would work as well?

A: There's nothing as general-purpose as logs.

You can record order by storing data in some other way (e.g. a b-tree) and storing sequence numbers with the data (Frangipani does this for meta-data, in addition to using logs).

You wouldn't have to worry about performance if you used a persistent storage system that was as fast for random updates as for sequential, for example battery-backed RAM. However, such systems are often more expensive and less robust than hard drives or SSDs.

For the write-ahead property, you could store a mini-log for each data record. However, it might then be time-consuming for the crash-recovery software to find the full set of incomplete mini-logs.

A different way to get crash-recoverable atomic operations is to prepare an entire new data structure in fresh storage, and then use a single commiting write to substitute it for the original data structure. This makes the most sense with tree- shaped data structures. The NetApp WAFL file system uses that idea:

https://atg.netapp.com/wp-content/uploads/2000/01/file-system-design.pdf

This arrangement may make it hard to support concurrent transactions.