550: Transactions

Main points

Transaction concept
Case study: Cedar

Even though this is a distributed systems class, I need to make sure you all understand the basic transaction concept, as its central to building workable distributed systems that can mask failures.   So today: transactions on one machine; Wednesday: transactions across multiple machines.

Simplest possible example:

Move file from one directory to another

In UNIX, a directory is just a data file (try on linux or a mac: "od –x ."). So it should be as simple as updating each data file, and storing it on disk. Right?

But can we make this work, despite failures? Let's assume for now the only thing going on is the directory move, so we don't need to worry about concurrent accesses.

Try #1:
Read A
Delete from A
Write new A
Read B
Add to B
Write new B

What happens if there is a failure at various points?   Say, after the new A is written, but before the new B is written.

Is there any other order that will work better?

Try #2:
Read B
Add to B
Write new B
Read A
Delete from A
Write new A

Uh, that's still a problem.

Here’s a possible solution: UNIX fsck. Special purpose code for all multi-object operations. For example, in the above, provided I know which sequence is being used, I can come along later, scan the entire file system, and diagnose and fix the problem.

Eg., if fail between the writes, then can look for a pattern (file in the inode list, but in no directory, assuming approach #1, or in both directories, in approach #2).

But it can get more complex: what if directory spans multiple file blocks? Might get halfway through the update, and end up with a mangled directory. What about other operations like deleting a file – it updates both the directory and other data structures.

So what we want is simple:

Atomicity: a set of changes to disk, that appear to happen together, or not at all. Regardless of when the failure occurs.

Very powerful idea, applicable to anything that needs to be very reliable. Example: doing a software installation – shouldn’t leave the system in a corrupted state, even if only get partway through the installation. Yet your portable insists on being plugged in to do an update; why? [most file systems provide atomicity for the data that organizes the file system, but not for user data.]

In particular, as we’ll see in assignment 3, it is a powerful idea for building fault tolerant distributed systems, since it gives very precise behavior of what to do when there is a failure – roll any changes back to the state of the world before the transaction started.

How do we accomplish atomicity? Requires that we can write a set of changes to disk, we can do that without them taking effect, and then we make a final write to disk that atomically switches from the old to the new state. The final write is a “commit”.

Of course, we might be able to do that by convention – e.g., if we store the directory name with the file, then we can update the directory atomically, and change the actual directory files in the background. Then we’d need to scan the disk for any changes that were partly done.

Maybe for changing the directory of a file we can do something ad hoc – e.g., e.g., each file has a pointer to its directory. Then writing the file sets the directory – the directory update is a hint that can travel along later. We’ll return to this idea later.

File systems have a lot of different types of operations, and can we make all of them atomic? Isn’t there a more general solution?

Two implementation techniques:

shadow file system. Let's assume that the file system is a tree (note: probably not true of your server!), so that you can find everything on disk by walking the tree. And we can garbage collect anything by looking for things not reachable from the tree.

Then we can make (any!) set of changes to the file system atomic, by making them to a shadow version of the file system, and then commit those changes by writing a new root, to a special location on disk.

(draw picture)

After a failure, can look at the special location to see which version of the file system we're supposed to use. (OK, in practice, since you are completely up a tree if the root block is corrupted, you write to a small array of root blocks that the file system scans on boot. The latest uncorrupted version is the file system.)

This is the basis of NetApp's storage system.

Upside? Can write the changes anywhere on disk – except for the root, which has to go in a specific place, everything else can go in any convenient free block.

Another upside is that you get versioning for free – old copies of the file system can be kept around with very low overhead.

Downside? Any individual small change may require a lot of updates, e.g., to all of the parent directories back to the root. So the key to reasonable performance is to be able to batch changes, and periodically commit them all in a bunch.

Approach 2 (taken in the Cedar paper): a "write-ahead log" (called a "redo log" in the paper). By a "log" I mean a part of the disk that you write once, but never update, like a ship's log. (I'll talk about how to reclaim the log in a second.)

[The Cedar paper also attempts to survive disk crashes, by replicating key data on the disk itself. At data center scale, disk crashes occur quite frequently. An easy solution is simply to mirror all data on two disks, so if one crashes, you can recover the data from the other disk. We'll talk about this more when we get to the Google File System paper.]

For write ahead logging to work, you need to write all your changes twice – first to a log, then to the file system. Once the data is in the log, you can write a "commit" to the log.

Once the commit is done, safe to copy the changes back to the file system, and reclaim the log for the next operation. (The Cedar paper has a more complex garbage collection scheme, but we'll keep things simple for now.)

What happens if there is a crash while you are writing to the log? Can ignore these. [Means that during recovery, we need to scan the log to see if the operation committed, before we do anything else. Anything that didn't get committed can be ignored.]

What happens if there is a crash after the commit, but before the changes were copied back? Read the log to re-apply any missing operations.

What happens if there is a crash after the log has been copied back to the disk? It might appear that the crash happened before the modifications were copied, so need to copy them over again – that is, we've turned the modifications into idempotent operations.

What happens if there is a crash during the recovery? Just restart the recovery from the beginning of the log.

Example: to move a file between two directories, write the two new versions of the directory to the log, and a commit. You can then update the cached copies of the data. Eventually, the cached copies will be written back to disk, in the normal way. Once they are on disk, we can reclaim the log.

If there's a failure, can ignore any operations that did not complete, and redo any operations that committed, but weren't copied back to memory.

Upside of write ahead logging? Once in the log, can apply changes in the background – as soon as change is in the log, you can tell the user that the change has been committed.

Can batch updates to the log, to reduce the overhead of synchronous commits. If you don't batch updates, Disk head movement: might need to ping back and forth between the log and where the data is being copieOriginally WAL for high end systems that could afford multiple disks. Why the theory was developed for databases (that is, banking), and not for programmers.

Downside? Have to do every disk write twice.

Variant: log-structure, where the log is the file system, so no need to copy changes back. Creates the issue of needing to be able to garbage collect the log, but most data on disk is not frequently modified.

I mentioned that most file systems do not provide atomicity for user data. How does an application get atomicity then? E.g., how do version control systems work? What happens if the system crashes during an update? Definitely don't want the version system to be corrupted.

Turn everything into whole file writes. Write a new copy with the update, then unlink the old copy.

A bit more on the 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

We've done atomic and durable. What if we need to be able to run multiple transactions at the same time? We will need to do this for distributed systems, since multiple clients => multiple transactions.

What if there are long-lived transactions?

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.

But if they are concurrent, should appear in one order or another. With failures, this can be pretty tricky to arrange!

If we allow reads during the transaction, then can see an inconsistent value (file in neither A nor B, or in both).

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.

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.

[Of course, 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: (a bit spiffier, and how we suggest you do it in assignment 3)

Another way to handle this is with optimistic concurrency control. When transaction starts, pick a (unique) virtual time for it to execute. All reads and writes are done consistent with respect to the versions of the data that were true at that virtual time.

To make this work, we can keep versions of data, so that if I update a data value, an earlier (in virtual time) transaction can continue to read the old data value if it needs it.

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.)

If a transaction reads a value written by an earlier transaction, then it can't commit until we know whether the earlier transaction commits.

But 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.

Using timestamps is just a generalization of checkpoints: grep is read only, so it logically can work on any (consistent) version of the database. But even if we have both reads/writes in a transaction, we can treat it as a logical checkpoint. It becomes “optimistic” if we allow the following transactions to start before we know whether they are affected by earlier ones – that’s not strictly necessary, but why not? We need to make sure that transactions don’t commit until all changes that they depend on have committed.

What about deadlock? In optimistic concurrency control, we never have deadlock! (similarly, in assignment 3: if a client with a read-write cached file is slow or crashes, the server can unilaterally revoke the read-write status, and simply abort any transaction in progress on the slow/crashed client.)

With two phase locking?

Can get deadlock, but with transactions, can always break a deadlock by aborting one of the waiting transactions. It will release any locks, revert back to the start, and retry.

Similarly, what if you get inside a file system operation, and 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>