490H: Replication

<everything gets simpler from here onward. It wouldn't do us much good if you needed to understand Paxos in order to use it. Rather, we can sweep it into a corner, use it to provide the basis for non-blocking failure recovery, and then on top of it, build simpler replication systems, and on top of those, mapreduce and bigtable and other cloud services, and on top of those, normal people clicking on the web, oblivious to any of this machinery.>

First, let me wrap up a few additional points about Paxos, that will be relevant for how it is used in practice.

The basic Paxos algorithm selects a single value, such that regardless of failures, once a majority has accepted the value, all further proposals will yield the same value. So we can think of the value being "committed", at the point when the majority accepts the value.

However, progress is not assured if multiple nodes are making proposals at the same time, because each could issue a prepare message in turn that would prevent any acceptor from accepting the other node's proposal. The natural solution is to always elect a leader first. Clients could then send their requests to the leader; the leader would then pick the order of operations, and have the Paxos replicas vote to concur on that order.

But electing a leader requires group consensus, so how can that work?

The idea is to elect a leader, as a "hint" – something that is usually true, but might be false, and doesn't lead to incorrect behavior when it is false.

For this, we can use any convenient algorithm to elect a leader. Let's assume that all nodes know about each other – there's a static list of nodes in the system. Then we can arbitrarily state that the lowest numbered node that is alive is the leader – so each node only needs to check periodically if a lower numbered node is alive. If so, they can't be the leader. If not, they are the leader.

Now this won't always work! But that's ok – we're safe even if there are multiple leaders.

The first thing a leader will do is to determine the state of the system: what decrees have been passed (accepted by a majority, where some alive node knows that it was accepted by a majority), and which decrees were started, but whose outcome is uncertain. So it will ping all the nodes to get their current state, which will (normally) cause any spurious leader to say: "Oh, you're the leader – great, then I'll stop being the leader."

Any uncertain decree, it will attempt to resolve. And then it can proceed, as long as a majority are up and responsive.

Clients talk to the leader; the leader decides on the order; the other Paxos nodes accept the leader's order (unless there are two leaders, which can get resolved quite quickly); the leader gets replies back from the Paxos nodes to verify the order was agreed to, and then the client gets the reply.

If one of the non-leader nodes fails, no problem – the leader just plods on with the remaining nodes. If the leader fails, the client will need to ask the new leader whether their operation finished – which can only be answered by determining the state of any chosen decrees.

Next, a common design pattern for using Paxos, is to use it for only one purpose: to set up a primary/backup system. The Paxos nodes decide on a temporary "primary" -- who is responsible for state for some period of time. The primary unilaterally decides on the order of operations, and tells a set of replicas before the operation is committed. (All of the logic in the book chapter now works – without the possibility of confusion because we've used Paxos to ensure that there is only a single primary at a time.) If there is a failure at the primary (or even if someone incorrectly believes that there is a failure at the primary), Paxos waits until the lease expires, and votes a new "primary", who can then read one of the backup copies to restore the state and continue. Otherwise, the primary renews the lease and continues. (Note this is the design pattern used in GFS, in the reading for Monday.)

If we are using hot standby replication, reads can go to any replica; writes go to the primary which then distributes them to the replicas before committing the change at the primary.

Alternatively, the primary can just send the log entries to the replicas. You can think of the replicas as each having a copy of the log. The replicas need to have the log on disk before the primary can commit, but they can process the log in the background.

Equally, we can think of the backups as passive disk storage. They get a copy of every disk write made at the primary, so that they can recover in exactly the same way that the primary would recover on its own.

In practice, a data center will run Paxos on a small number of nodes, and they will be responsible for selecting the various primaries for all of the clusters in the data center.

The chapter makes the point of distinguishing MTTF and MTTR – availability is affected by the product of these. So we can improve system availability either by reducing the rate at which things fail (e.g., by building better power systems, or by writing simpler software), or reducing the time to recover from those failures (using

hot standby replication, or even by using write ahead logging instead of full disk scan recovery schemes as in traditional UNIX and Windows file systems.)

How does Paxos and primary/backup interact with the project? To answer that, I need to walk through two possible designs for assignments 3, 4 and 5.

Suppose we wanted to use two phase locking. Then the cache coherence system we had from assignment 2 would be the same, except that we'd have a few more states. Read-only, write-only, read-lock, write-lock, and invalid.

What is the state transition graph? When we read data in the transaction, we move the read-only data into read-lock – preventing any competing transaction from writing the data. Commit moves the data back into read-only. If the data is invalid when it is read in the transaction, then you first move it to read-only, then into read-lock. Similarly for write-only: writing data in a transaction would get the data as write-only in the local cache, and then write-locked, to prevent any competing transaction from reading the data.

This is blocking, in that a server failure, or a client failure, could cause a client to stall until the failed node recovered, but that's ok in assignment 3.

And you might have deadlock, but since you are able to write your own transactions in assignment 5, you can use lock ordering to eliminate deadlock.

How can we use paxos to make this solution non-blocking?

We need there to be a single server at a time – so we can use Paxos to name the server for a period of time, and replace it when it fails.

Distinction between hard and soft state – if the state is lost can we recover it? Then it is soft. Is the server state hard or soft? (soft – we can recover the callback state from the other clients). So when the server is replaced, we can recover the state and keep operating.

How about the client state? The client log and disk contents are hard state – so these need to be replicated, using any of the primary/backup schemes outlined above. If the client fails, the server can ask one of the replicas to take over for the failed node.

OK, but most of you are probably using optimistic concurrency control, and several of you have observed that there's a bit of a disconnect between transactions and cache coherence in that case. As before, let's talk about assignment 3 first, then add Paxos in later.

I think the easiest way to navigate this is to (temporarily) imagine a system entirely without caching.

Each client would fetch the latest copy of whatever files it needed for a transaction, and then send back any updates to the server at the commit.

The server can then decide whether a particular transaction can be committed (did any of the files it read change in the meantime?), and provide the latest copy of any committed file back to any client that asks for it.

How might we modify this solution to add caching? One way to do this would be to use write-through cache coherence.

A client can cache read-only versions of a file, and the server can keep track of who has a copy. When a client commits a change, it sends the change to the server, which then invalidates the other copies. If there is a transaction in progress at the client using the invalidated copy, it is clear that transaction will need to be aborted, and restarted with the new version.

Can we make this work with write-back cache coherence? If we don't mind blocking on client failures, yes! When a client commits a file modification, it sends the server a notice that the file has changed, but it doesn't need to provide the actual file contents.

The server then notifies all the other clients that there is a new version, which they can then fetch from the client in the normal way.

If the client commits a transaction, involving files for which it has write-exclusive access, it doesn't even need to tell the server when it commits the transaction, since it is guaranteed to have serializability with respect to all other concurrent transactions occurring at the other clients.

OK, then: how do we make this non-blocking?

If we have no caching, it is pretty easy: clients send their proposed transactions to a set of servers. Once the data is on disk, they send the commit to the leader. The leader proposes a consistent order for the transactions, and the other nodes vote to accept that order.

Alternately, we can use Paxos to elect a single server for a period of time, to serve as the primary. That server would need to replicate its disk storage across a set of backups, who would take over if Paxos selected a new primary.

In this world, we can easily accommodate write-through cache coherence – the server maintains callback state, which is reconstructed (by the new server selected by Paxos after the server lease expires) by polling the clients if the server fails.

And this can accommodate write-back cache coherence as well. Whichever client has the latest copy, has to replicate that copy out to a set of backups before its changes can be committed.

A client is granted a lease on its write-exclusive data by the server (which was granted a lease by Paxos). If the client fails, the server nominates another node to take over from the server (after the lease expires), and that replacement can read the client's replicated logs to reconstruct the latest version of data.

Finally, partitioning vs. replication. We might be concerned that the server becomes a bottleneck – even with Paxos nominating a server, and the server pushing as much work as possible off to the clients, maybe the amount of work being done by the server is too much. (especially if it is involved in every transaction.)

We can then consider partitioning the work between multiple servers. This is easiest if the servers can be assigned disjoint parts of the state space – if transactions at one server never touch the data at the other server, or if the transactions at each server don't need to be serialized with respect to one another.

If we do need serializability across partitions, then we need to run something like two phase commit to provide ordering. Two phase commit is normally blocking, but if each of the partitions is managed using Paxos, then any failure of the coordinator can be masked by the backup server – which can take over and complete the transaction.

One can think of write-back cache coherence as a kind of dynamic partitioning – items are assigned to a client, and if no other client needs those items, the writer can manage those transactions entirely without central involvement.