Paxos applied

<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.>

Paxos provides a highly available, redundant log of events

(events can be transactions, storage accesses, who is primary)

Replicated state machines: supply the same sequence of events to a deterministic state machine -> get same answer

With RSM, ok to consult any participant to fetch the log
(to add an event to the log, need to run paxos or 2pc)

Can also use Paxos to manage sharded databases, but that's a bit more complex so I'll cover that later.

Bottom line: Can use Paxos to build all sorts of interesting cloud services.

There are a bunch of practical engineering issues, which you'll run into when you do project 3 – I realize that's a ways off!

0)using Paxos

Paxos requires a configuration step: list of nodes participating in the election

value is chosen if accepted by a majority – a majority of the original list of nodes (NOT of the set who are currently alive!)

later: how we change the set of nodes; for now I'll assume we know them

also: let's assume that every node serves as proposer/acceptor/learner

a client talks to the Paxos group as if it is talking to one server; it sends a request to any participant and they collectively vote on the sequence of client requests; reply back to the client when the action is committed (chosen, that is, accepted by a majority, and learned)

1) picking proposal #'s (for voting on which event goes into a particular slot in the sequence): I said the proposal # needs to be unique across all proposals, but doesn't that require consensus?

Proposal # = max proposal # + 1 ???

Rather: ith node making jth proposal among k nodes:

Proposal # = j*k + i

2) picking a leader

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.

Paxos algorithm may not make progress if there are two simultaneous proposers; also won't make progress if there are zero proposers

if multiple nodes are making proposals at the same time, 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.

However, can't guarantee exactly one proposer! (without running paxos to elect the proposer!)

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.

But we can engineer the system to almost always have one proposer: out of k nodes, pick lowest node that is up; if I've seen a recent message from a node with a lower #, stop proposing

Then various things you might do:

If you think you are the lowest #, broadcast a ping to everyone to let everyone know you are up

If hear a proposal from someone, when you've recently heard a proposal from someone with a lower #, ignore the higher # (they are probably just misinformed)

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

3) use of stable storage

proposer needs to keep track of last proposal # used (since can't reissue a proposal #)

acceptor needs to keep track of response to prepare request (to ensure don't accept any lower #'ed proposals)

acceptor needs to keep track of response to accept request (what value have I accepted)

learner needs to store chosen value

=> 3 + k disk writes

use flash!

And/or when node fails, don't let it recover! Then we only need a disk write for the learner, and that can be batched. Does mean we need some way to remove a node permanently, and to add a new node into the system permanently (that is, we can use a different way to recover after a node failure – one that requires the vote of all of the participants to exclude a node or to let the node back into the system – that vote gives an opportunity to erase any pending state)

4) from one paxos to many

Algorithm is highly available, redundant log of one event -> how do we get to a highly available sequence of events
Terminology: instance of Paxos refers to one slot in the sequence of events

a) If no failure (or late messages), easy: original leader (the one with the lowest ID #) can make a proposal for each instance in turn, send a prepare, and get it accepted

higher performance version: ok to run multiple instances of Paxos in parallel.
For example: two clients each request a transaction, that are commutative

Leader chooses the order, e.g., request A is instance 57, request B is instance 58

Send out the prepare and accept requests for each at the same time

- ⇨ B might be chosen "before" A! (e.g., with packet loss) -- ok to tell client that B is complete
- ⇨ But assuming A is eventually chosen, then A is put in the sequence at 57, and B at 58

Even higher performance version: Leader sends a single prepare message for all possible instances first (I'm about to send proposal #i), then only the accept request needs to be sent for each instance

b) If a non-leader fails, easy: leader keeps on chugging with the remainder

When the non-leader recovers, it uses its log to prevent it from responding incorrectly to prepare/accept requests. It can also consult any other node in order to catch up with the log.

c) if leader fails, a bit more complex: if haven't heard from the leader recently, timeout and broadcast a request to see who is the lowest #'ed remaining node.
If leader says: hey, I'm still here, ok, done.

Otherwise, lowest remaining # (new leader) needs to pick up where the leader left off.

The first thing a new 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."

Some instances may:

have a chosen value that has been learned by someone who is up; go poll everyone to find out what has been learned (e.g., B at 58)

have a chosen value that hasn't been learned; go poll everyone to see their latest accepted value

have a proposal that has been accepted by some nodes, but without enough to know if the proposal has been chosen yet: run a new proposal (with a larger #!) to choose a value consistent with prior accepts

no proposal accepted yet – ok to fill with any proposal that is commutative with already accepted instances. It might be that the client was told B was chosen at time slot 58, so its not linearizable to put a later request to do C in slot 57. It is always safe to insert a no-op into the sequence to fill a gap.

NOTE: recovery process must be idempotent – that is, if new leader fails, next leader just picks up and does the recovery process again, no matter how much progress the failed leader had made.

When old leader recovers, can start back in (if you want to keep constraint that leader is always lowest #'ed node), taking over from the current leader in the same way as if the current leader failed

d) one final case: if leader is slow, but hasn't failed

then there could be two leaders making proposals at the same time. Paxos guarantees that this is still correct, even if it might stall progress until its sorted out.

Monday:
Continuing with Paxos.
Draw picture of clients communicating with Paxos group

<insert: I answered a question a bit confusingly last time>

What about reads?

a) if the client is ok with an old read, then can answer from the log – the read can be satisfied by any
b) if the client wants a guarantee that no one has updated the value in the meantime (that its reading the latest value), then the read needs to go to a majority
c) most systems handle this differently: they give the leader a lease. During the lease, the master is the only leader, and therefore, any reader can trust that the master has the latest version. This means the system can be offline for a short period if the leader fails, as everyone will need to wait for the lease to expire before proceeding.

5) How does a leader failure affect clients? Client request can be in various states:

Chosen and learned and returned back to the client – done
Chosen and learned and not returned to the client – client asks new leader for status
Chosen and but not learned – client asks new leader for status, and new leader replies after the result has been learned
Accepted by some acceptors, but possibly chosen or possibly not chosen – client asks new leader, and new leader attempts to resolve request, possibly inserting a no op in previous items and asking client to retry

If client request can be proven to have not been chosen (e.g., no live nodes have accepted the proposal to slot that request at any instance, and a majority of nodes are live), then new leader can reply to client: just try again. Client request gets assigned a new tentative instance #, they vote, and agree on its position. That's consistent with linearizability, since result hadn't been conveyed back to the client.

But! Needs to do this in a way that guarantees no sequence of failures will cause client request to be done twice! E.g., need to elect a no-op at 57 to guarantee that if old leader recovers, it won't try to vote in A at 57. That makes it safe to put A at some new instance #.

6) changing the set of voters

a node crashes permanently and needs to be replaced
a node finds that its file system has been corrupted
a new node needs to be added to the set
performance optimization: we don't need to save prepare/accept state on disk if we never allow a node to recover, but only allow replacement

conceptually simple: just put an event in the sequence to change the set of voters; applies to all following events

this "view change" event must be chosen by a majority of the PREVIOUS set of acceptors. This event is NOT commutative with respect to other instances, so you can't run parallel instances

Thus, if two of k nodes fail, you can potentially vote among the remainder that you will no longer accept proposals from the missing nodes. Then for new votes, you will only need to get a majority of the remainder. The failed nodes might not really be failed of course – but they won't be able to form a majority (of the original set of nodes), since that would overlap at least with one of the nodes that approved the view change.

7) intermittent network outages

Sometimes a particular connection may go up and down quickly – so that some nodes may think a node is down while it isn't.

If this happens to a regular node, it falls behind and has to catch up. (It can keep accepting proposals, but if enough nodes fail to keep up, you could have to redo a bunch of work to re-learn chosen proposals.

If this happens to the leader, then a new leader will be chosen, but what happens when the old leader comes back online. Most systems try to avoid leader swaps, e.g., once someone becomes a leader, they stay the leader until they fail. Again, leader election doesn't have to be perfect – since it doesn't violate correctness to have two leaders.

Another thing that can happen is the network becomes partially disconnected – e.g., A can talk to B and B can talk to C, but A can't talk to C. Thus, A thinks C is down, and vice versa, but B thinks both A and C are up.

This can lead to problems when choosing a leader, e.g., if both A and C think they should be the leader.

Solution? Some careful coding! E.g., exchanging information among nodes about who they can communicate with (don't trust the network to always be fully connected).

8) snapshots and garbage collection

as defined, the Paxos log grows without bound (and therefore you would eventually run out of disk space!)

How can we truncate the early entries? We need to take a snapshot – store the state of the system as of some event # in the log. Because this is a replicated state machine, the snapshot can be computed directly from the log – we don't need any coordination. When the snapshot is safely on disk of all of the participants, we can tell the learners that they can reclaim the log up through that snapshot. On recovery, we start with the snapshot, and go forward from there.

9) scalable storage
mentioned earlier that we can use Paxos to manage a primary/backup system. 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.

Each primary will be responsible for a portion of the total storage, e.g., a single shard of the overall data.

The Paxos nodes decide on a temporary "primary" for each shard -- 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 to store the result, and when they reply, the primary can reply back to the client. (Despite failures, there's no 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.

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.

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. (with some delay)

10) paxos + 2pc

ok, operations on each shard are serializable, but how do we order transactions that need to access/modify data across multiple shards?

For that, we need to layer 2pc across shards.

Wait! Isn't that blocking?

Well no: each shard is highly available – if the primary fails, we have paxos vote in a new primary. Then even if the 2pc coordinator (the primary of one of the shards participating in the transaction) – even if it fails, one of the other nodes in that shard can take over as the new primary, and complete the transaction.

11) Byzantine Paxos

Paxos (as described) depends on every node following the protocol correctly. What happens if a node goes rogue? E.g., due to a software bug or possibly a malicious attack?

Sends a prepare message, then an accept request without waiting for the prepares (or without picking the value to be the one of the highest proposal with any acceptor)?

Accepts a proposal even when it promised not to?

Result might be a subtly corrupted database that could be very difficult to unwind back to a correct state. Can we design a Paxos that works despite some # of misbehaving nodes?

Turns out yes, and the protocol follows Paxos with a twist. If there are no more than f Byzantine nodes, we can still make progress if we have 2f+1 correct nodes – that is, we need 3f+1 instead of 2f+1 nodes.

The key idea is to distrust anything said by only f nodes – but if something is said by f+1 nodes, then a working node can trust it.

For example, to learn if a value has been chosen – in normal Paxos, we can learn by asking any other node if the value has been chosen. But with Byzantine nodes, the node could be lying or mistaken! So instead, we have the acceptors each sign a certificate, saying that they accepted the <proposal, value>. Then the learner can prove its heard enough accepts, to anyone who asks.

How many acceptors do we need? A value is chosen if it is accepted by a majority of the working nodes – so we need 2f+1 acceptors, to ensure that we have f+1 correct acceptors. Note that this only tolerates f failures – we can make progress only with f or fewer broken or malicious nodes. (To work in the presence of f broken/stopped nodes and f malicious nodes, you need more – 4f+1 nodes).

You can then work your way back through the protocol along these lines: how does an acceptor rely on the correctness of the accept request? (That is, that it correctly represented the value of the highest #'ed proposal that was accepted?)