Chan and Golab:
Paxos

Introduction

Paxos is a possible solution to the consensus problem. The core of the problem is about "which of these actions should I execute first"? Consistency is all about whether distinct servers can agree on what is the "true" execution.

Paxos is good because despite some theoretical limitations, it works really well in practice. It gives us safety (we'll never decide on something that wasn't proposed) at the cost of occasionally sacrificing liveness (might take a long time, theoretically possibly ∞ time to decide).

Let's start with some assumptions.

1) No Byzantine failures
2) Processes are async, can fail by crashing, and recover by restarting
3) Processes have stable storage to help recovery
4) Any process can message any other
5) Unbounded communication delay
6) Messages can be lost or reordered, but not corrupted
7) Progress on the protocol is only guaranteed when both the processes and the network are reliable and timely.

Each process plays one or more roles in the protocol.

Client - gives a value to propose to the proposer
(a value simply means a state transition to execute)

Acceptor - an acceptor process "holds" the decided value
- a quorum of acceptors must accept a value for the entire protocol to decide a value
- any two quorums must overlap

Proposer - proposes values to acceptors

Learner - only come into play after a value is decided by the protocol
- does an external-facing action based on the decided value

Leader - special type of proposer
- needed to progress the protocol
- decided by non-Paxos related algorithm

<u>How it Works</u>

Paxos works in two distinct phases. The first phase tries to determine if a value was decided on before the current round began.

The second phase is used to determine if a new value is decided on. These two phases are repeated indefinitely until consensus is reached.

Each proposal is given a unique proposal number, unique to a specific process (ie. process 1 and process 2 can't both make a proposal numbered 5). These numbers increase monotonically.

Phase 1A: Prepare

1) Leader creates proposal N, with N larger than anything it's used before
2) Leader then sends prepare (N) to a quorum of acceptors, chosen by the leader itself

Phase 1B: Promise

1) At the acceptor, if N > any proposal number the acceptor has received, it sends back promise () with the most recently accepted (not decided!) value and that proposal's proposal number
2) Otherwise, acceptor sends back NAck saying to try again with a higher proposal number

Phase 2A: Accept Request

1) If the leader receives promise (n, v) from a quorum of acceptors, the leader sets its own value to the ~~highest~~ value associated with the highest n it got back
2) If no acceptor accepted anything previously, the leader can set the value to whatever it wants
3) Proposer sends AcceptRequest() to a quorum of Acceptors with the chosen value for its proposal

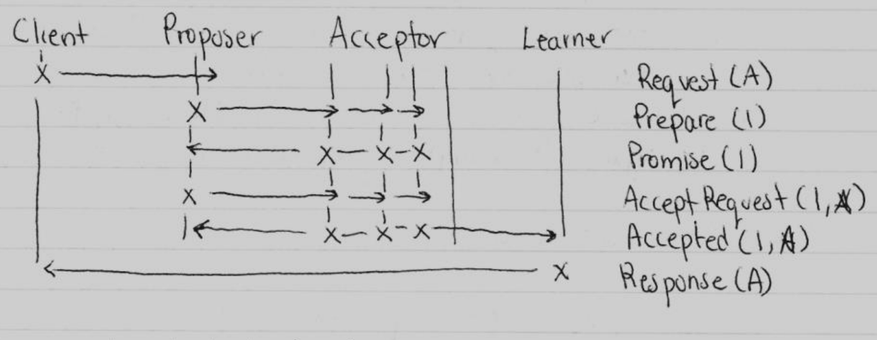
Phase 2B: Accepted

1) If an acceptor receives AcceptRequest(N, v) it accepts if its Promised n is smaller than N. It holds them in memory, then sends Accepted(N, v) to the proposer and all learners.

2) If it cannot accept the proposal, it sends Nack (n), where n is its saved proposal number

Okay, so we've covered the steps, let's see how it works in practice, first with no failures. Remember that only when a majority of Acceptors accept a value, is the protocol considered "done" (in the absence of new proposals.

ex. We'll start fresh from the very beginning. No acceptors have previously accepted anything.



Let's walk through it step by step.

First, the client has a value to propose, A. It makes a request to the system to accept A.

The proposer then chooses a quorum of acceptors (Acceptor 1, 2, and 3 out of 4), and sends its first Prepare with the proposal number 1.

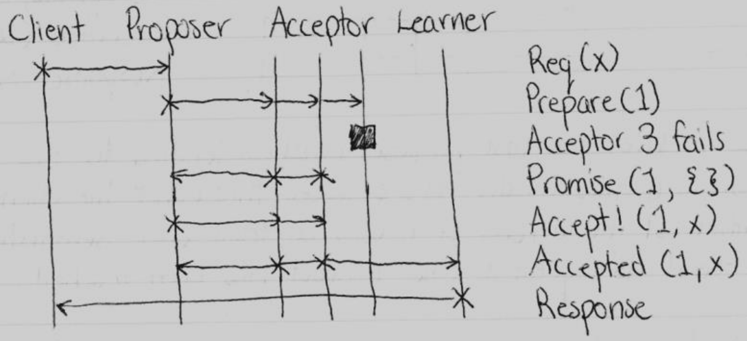
The acceptors have not previously promised anyone anything, so they gladly say ok, and promise not to accept prepares or proposals with round number less than 1.

Next, since the proposer didn't get a higher proposal number back, it sets its number and value to (1, A), and sends the accept request to the same three acceptors.

The acceptors have promised to accept prepares/proposals equal to or larger than 1, so all three accept the value given. These are forwarded to the Learner.

The Learner then performs some operation and responds to the client, saying its A has been accepted.

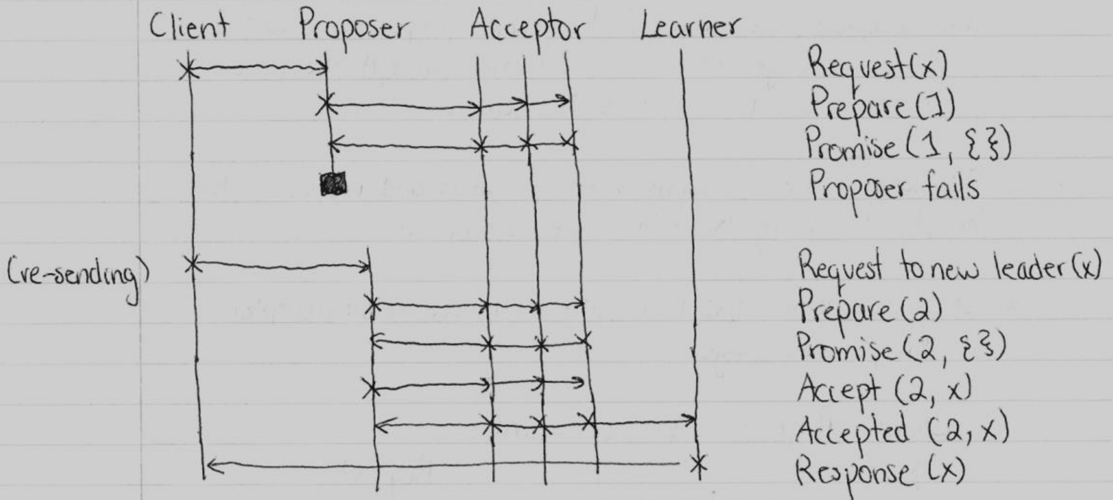
ex. Let's do another, this time with the failure of an acceptor at the promise stage.



Here, the request and prepare phases are the same. At some point after the prepare has been received and before the promise is sent out, Acceptor 3 fails.

However, since the proposer still receives a quorum of promises, and those promises don't have previously accepted values, the proposer is free to propose x and the protocol continues as normal.

ex. So what happens when the proposer dies? Usually, a second process will be elected the leader, and the protocol will restart.



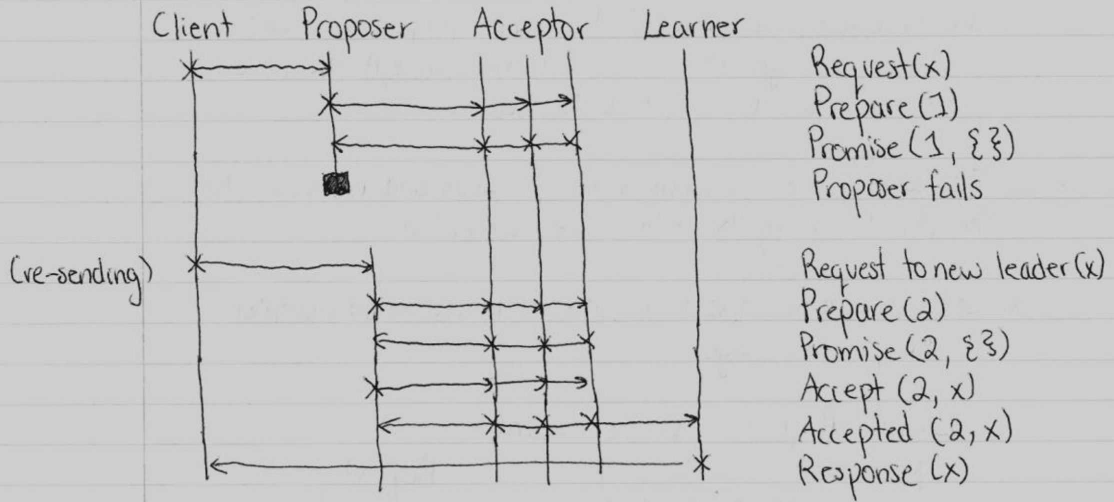
In the middle, the first proposer crashes, leading to the election of a new leader. It's assumed that either the client is notified of the failure, or it uses at-least-once semantics to re-send its request if some timeout has been reached.

After that, though, the protocol can continue as normal.

However, Paxos is built upon the assumption that because processes have stable storage, instead of crashing and then stopping, processes can crash and then recover.

So what happens if the first proposer recovers, and thinks that it's the leader?

ex. So what happens when the proposer dies? Usually, a second process will be elected the leader, and the protocol will restart.



In the middle, the first proposer crashes, leading to the election of a new leader. It's assumed that either the client is notified of the failure, or it uses at-least-once semantics to re-send its request if some timeout has been reached.

After that, though, the protocol can continue as normal.

However, Paxos is built upon the assumption that because processes have stable storage, instead of crashing and then stopping, processes can crash and then recover.

So what happens if the first proposer recovers, and thinks that it's the leader?

The new leader enters the accept phase - and asks the acceptors to accept V_b with proposal number 2. But they've already agreed to ignore proposals ≤ 3! So they send back a NACK(3), saying try again, we need a [illegible] proposal number higher than 3. And so it does just that - starts the protocol with 4, and receives some promises.

This will, in turn, "cut off" the old leader's proposal, and tell the old leader he needs a proposal number higher than 5.

If this process keeps going, without external interference, it could go on forever. This is why Paxos is only guaranteed to converge when there is one leader. Generally, though, once the network settles back down, the two proposers can see that the other exists, and will agree to let one go first, which breaks the potentially infinite loop.

There's one last thing to touch on - whether acceptors can accept a value DISTINCT from the one it has already accepted.

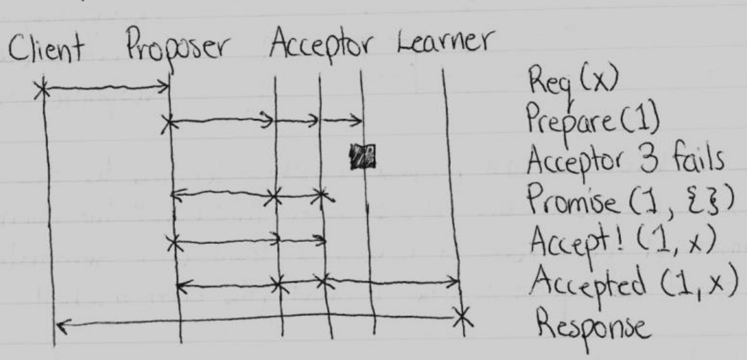
The answer is yes, and it's based upon certain failures occurring, and multiple clients requesting different values.

Next, since the proposer didn't get a higher proposal number back, it sets its number and value to (1, A), and sends the accept request to the same three acceptors.

The acceptors have promised to accept prepares/proposals equal to or larger than 1, so all three accept the value given. These are forwarded to the Learner.

The Learner then performs some operation and responds to the client, saying its A has been accepted.

ex. Let's do another, this time with the failure of an acceptor at the promise stage.



Here, the request and prepare phases are the same. At some point after the prepare has been received and before the promise is sent out, Acceptor 3 fails.

However, since the proposer still receives a quorum of promises, and those promises don't have previously accepted values, the proposer is free to propose x and the protocol continues as normal.