Paxos By Harry Brundage http://harry.me/blog/2014/12/27/neat-algorithms-paxos/ Intuitive Explanation

PAXOS is an algorithm to solve the consensus problem. Honest-to-goodness real-life implementations of Paxos can be found at the heart of world class software like Cassandra, Google's magnificent Spanner database, and also their distributed locking service Chubby. A system governed by Paxos is usually talked about in terms of the value, or state, it tracks. The system is built to allow many processes to store and report this value even if some fail, which is handy for building highly available and strongly consistent systems. To restate, a majority of the members of the system must agree that a particular value is in fact "the one true" value to then report it as such. Conversely, it means that one rogue process which has an out of date idea of the world can't report something that isn't "the one true" thing.

Let's get some definitions out of the way for upcoming explanation:

- A `process` is one of the computers in the system. Lots of people use the word replica or node for this as well.
- A `client` is a computer who isn't a member of the system, but is asking the system what the value is, or asking the system to take on a new value.

Paxos is only a small piece of building a distributed database: it only implements the process to write exactly one new thing to the system. Processes governed by an instance of Paxos can either fail, and not learn anything, or by the end of it have a majority having learned the same value such that there is consensus. Paxos doesn't really tell us how to use this to build a database or anything like that, it is just the process which governs the individual communications between nodes as they execute one instance of deciding on one new value. So, for our purposes here, the thing we build with Paxos is a datumbase which can store exactly one value, and only once, such that you can't change it after you've set it the first time.

The read guts

To read a value from the basic Paxos system, a client asks all the processes in the system what they have stored for the current value, and then takes the value that the majority of the processes in the system hold. If there is no majority or if not enough processes respond, the read fails. To the left you can see a client asking the nodes what their value is, and them returning the value to the client. When the client gets a majority of responses agreeing on a value, it has successfully read it and keeps it handy.

This is weird compared to single node systems. In both places, the client needs to make an observation on the system to determine the state, but in non-distributed systems like MySQL or one memcached process, the software only needs to ask the one canonical place where that state is stored. In simple Paxos, the client needs to observe the state the same way, but there is no canonical place where it is stored. It needs to ask all the members, so that it can be sure that there is actually only one value reported, and that it is in fact held by a majority of nodes. If the client just asked one node, it could be asking a process which is out of date, and get the "wrong" value. Processes can be out of date for all sorts of reasons: messages to them might have been dropped by unreliable networks, they might have failed and recovered with an out of date state, or the algorithm could still be underway and the process could have just not gotten it's messages quite yet. It is important to note that this is "naïve" Paxos: there are much better ways of doing reads when implementing a system using Paxos that don't require contacting every node for every read, but they extend beyond the original Paxos algorithm.

The write guts

Let's examine what Paxos makes our cluster of processes do when a client asks that a new value be written. The following procedure is all to get only one value written. Eventually we can use this process as a primitive to allow many values to be set one after another, but the basic Paxos algorithm governs the flow for the writing of just one new value, which is then repeated to make the thing actually useful.

The process starts with a client of the Paxos governed system asks that a new value be set. The client here shows up as the red circle, and the processes show up as the teal circles. Paxos makes a guarantee that clients can send their write requests to any member of the Paxos cluster, so for the demos here the client picks one of the processes at random. This property is important and neat: it means that there is no single point of failure, which means our Paxos governed system can continue to be online (and useful) when *any* node goes down for whatever unfortunate yet unavoidable reason. If we designated one particular node as "the proposer", or "the master" or what have you, then the whole system would grind to a halt if that node failed.

When this write request is received, the Paxos process that receives the write request "proposes" this new value to the system. "Proposition" is in fact a formalized idea in Paxos: proposals to a system governed by Paxos can succeed or fail, and are a required step to ensure consensus is maintained. This proposal is sent to the whole system by way of a `prepare` message from the process the client contacted to all the other processes it knows of.

Sequence Numbers

This `prepare` message holds inside it the value being proposed, as well as what's called a *sequence number* inside it. The sequence number is generated by the proposing process, and it declares that the receiving process should prepare to accept a proposal with that sequence number. This sequence number is key: it allows processes to differentiate between newer and older proposals. If two processes are trying to get a value set, Paxos says that value proposed last should take precedence, so this lets processes figure out which one is last, and thus who is trying to set the most recent value.

These receiving processes are able to make a critical check in the system: is the sequence number on an incoming `prepare` message the highest I've ever seen? If it is, then cool, I can prepare to accept this incoming value, and disregard any others I have heard of before. You can see this happening to in the demo on the right: the client proposes a new value every so often to one process, that process sends `prepare` messages to the other processes, and then those processes note that these successively higher sequence numbers trump the older ones, and let go of those old proposals.

This little ordering idea is what lets any member of the system issue proposal to avoid the single point of failure associated with a designated "proposer" node mentioned above. Without this ordering, members of the Paxos system would have no way to figure out which proposal is the one they should prepare to accept with confidence.

We could imagine a different consensus algorithm which didn't do this step of sending a first message to ask the other processes to make sure the value trying to be set is the most recent one. Although being way simpler, this would no longer satisfy the consensus algorithm safety requirements. If two processes started proposing different values right around the same time (like in the demos below), the universe could conspire against us and align the packets such that each dueling proposer convinces one half the processes to accept their own maybe-right-maybe-wrong value. The system could end up in a stalemate! There would exist two evenly sized groups having staged different value, which would lead to no value being accepted by a majority group. This stalemate is avoided by the first Paxos message exchange with sequence numbers that allow the processes to all resolve which proposal they should accept. With Paxos' sequence numbers, one of the dueling proposals would have a lower number than the other, and thus upon proposal receipt processes will have a way to unambiguously pick the most recent one. They'd either get the higher number one first, and later receive the lower number one and reject it, or they'd get higher numbered one second and thus replace the lower numbered

one with it. Paxos solves the problem of consensus over time by taking hold of time itself with sequence numbers to apply temporal precedence.

Side note: it's important that no two proposers ever use the same sequence number, and that they are sortable, so that they truly reference only one proposal, and precedence between proposals can be decided using a simple comparison. When implementing Paxos, these globally unique and sortable sequence numbers are usually derivatives of precise system time and node number in the cluster so they grow over time and are never the same.

Promises

So, after the proposing process has sent out it's proposal, the processes check the proposal's sequence number against the highest they've ever seen, and if it is the highest, they can make a promise to not accept any proposals older than this new threshold sequence number. This promise is returned as a message sent from the promising process to the one that is proposing a new value, as thing a `promise` message. This gives the proposing process the information it needs to count how many processes have sent their promises, and thus the basis to establish if it has reached a majority or not. If a majority of processes have agreed to accept this proposal or a higher sequenced one, the proposing process can know it "has the floor", so to speak, and that progress is possible in the algorithm. If for whatever reason the proposer can't extract a majority of promises from the other processes, progress is impossible, since consensus couldn't be reached, so the proposal is aborted and the client is informed that the write failed.

To decide if a proposal has extracted enough promises, proposers simply count the number of `promise` messages they receive and compare against the total number of processes in the system. "Enough" promises here means promises from a majority (N/2 + 1) of processes in the system are received before a certain timeout. The simplest reason for this might be that more than half of the processes in the system have failed completely, so they wouldn't return `promise` messages ever. This means Paxos could never get the proposed value committed a majority of processes, and thus could never satisfy the majority requirements in the read algorithm described above, and thus couldn't reach consensus, so the proposal should be aborted. Other failure modes which would prevent a majority of promises being returned include things like a network partition preventing a proposer from contacting enough nodes, or more interestingly, a competing proposal having extracted `promise`s with a higher sequence number already.

Acceptance

Once a proposer has extracted promises from a majority of other processes, it asks the promising processes to "accept" the value they promised to before. This is the "commit" phase of the algorithm where progress is actually made. If there are no dueling proposals, failures, or partitions, then the proposal will be accepted by all nodes, and Paxos is complete! You can see this is the demo to the right when the second round of messages from the proposer, called the `accept` messages, cause the promised values to be taken on (sucked in) by all processes.

Acceptance of a particular process can fail however: if enough processes fail right after replying with a `promise` message, but before they receive the `accept` message, then the acceptance could only happen on a minority of nodes instead of a majority. In this case, the Paxos round is now in a weird state where some processes have accepted a value, but not all. This state, while undesirable, is actually "consistent" due to the read logic described far above: a client trying to read from the system must receive agreement from a majority of nodes on what the value actually is, so if it managed to contact all the nodes, different, conflicting values would be reported by different minorities of nodes. This would cause the read to fail, which sucks, but Paxos has remained consistent, and hasn't allowed a write to take place without consensus. This bad state is often corrected in real implementations by either repeating the accept phase to get more nodes and eventually a majority.

Dueling Proposals

Acceptance can also fail because of dueling proposals: the promise that promisers reply with is a contract to accept proposals with that proposal's sequence number, or higher. This means that a second proposal could come around after a first with a higher sequence number, and extract new promises from all the processes where they would no longer accept the first, earlier proposal. The first proposer however might not find out about this second proposal, and continue happily along in the Paxos algorithm, and send out it's `accept` messages. Upon receipt of those `accept` messages, the promising processes would note the lower sequence number than their second promises to the second proposer, and simply reject the `accept` messages. This is correct: the first proposal sadly hasn't made progress, but no value was accepted without consensus, and Paxos remains consistent. This kind of situation could easily arise if clients cause two different processes to start proposals at the same time, which you can see above.

The failure case here gets even more complicated if the second proposal comes around *after* acceptance has happened on some nodes for an earlier proposal. This is a Danger Zone for Paxos: if different processes have accepted different values, and especially if the groups of processes change which value they have accepted over time, reads of the system could return different values at different times for the same round of Paxos! That violates the consensus algorithm safety property that only one value can be reported by the system, so let's examine how Paxos handles successive proposals and acceptances.

Let's say that a second proposer recovers from a network partition and tries to propose a new value after a first proposer has already proposed a value, and already had that value accepted by a majority of processes. Paxos has "completed", in that if the read algorithm was run, the first proposer's value would be reported as the value for the system. That said, this second proposer is allowed by Paxos despite the completed state, because it declares all stages repeatable to allow for failure correction. So, this second proposer can run, but it must not change what value has been accepted, to keep consensus in tact as described above.

To prevent these later proposals from changing the accepted value, Paxos adds a little doohickey to prevents proposal from having a different value than the first one. If any processes have already accepted a value, Paxos forces *any* proposal that comes after that to also have that same already-accepted value. Freaky, but this maintains consensus, because now the value can never change after acceptance has started. The way this is implemented is that the `promise` messages returned by processes which have already accepted a value also carry along what value has already been accepted, and can tell the proposer about old values that have been accepted. The proposer can then detect if a majority of nodes have already accepted an old value, and change it's proposal to match that value, or not run it at all. In a way, Paxos piggybacks a read operation on top of the promise phase to make sure that a proposal is in fact free to change the value of the system.

Datumbase

All this procedure accomplishes one thing: one durable write. Paxos itself has many variants that make it faster, introduce the ideas of masters, sacrifice pure fault tolerance for more speed, and tonnes of layers built on top which use it as a primitive to implement an actual database. An extremely interesting description of how to do this can be found in the Paxos Made Live paper listed below, but for us, this is the end.

By Harry Brundage http://harry.me/blog/2014/12/27/neat-algorithms-paxos/