Paxos

A Preprint

Victor Yodaiken*

October 20, 2019

Abstract

The Paxos protocol for distributed consensus.

Keywords automata, state, recursion, consensus, network

1 Introduction

Paxos[1] is notoriously hard to understand because of the number of moving parts and the confusing descriptions of the algorithm, but it is fundamentally pretty simple. The problem Paxos is intended to solve is ensuring that a collection of network nodes (sites) that communicate only by message can come to a consensus about some value even though some messages may be lost or delayed and nodes can fail. A trivial "brute force" algorithm works well in these kinds of networks with a single node that can propose values, but if that node fails, the system fails. The original Paxos algorithm (abandoned 3/4 through the "Paxos Made Simple" paper) has a multiple "proposers" that compete for majority consensus on some value. Normally, these kinds of algorithms are considered "safe" if at most one proposer can win, but the ingenious innovation in Original Paxos allows any number of proposers to win as long as the proposers come to a consensus about the proposed value.

2 Original Paxos Safe

The network model includes a set of network nodes and a set of messages that pass between nodes. Messages can be lost or delayed, but if site n has received message m then some node n' must have transmitted m in a previous state (no spurious messages). Suppose we have a set A of acceptor nodes and a non-empty, non-intersecting set P of proposers. A proposal is a triple $m = (p, q, v)$ where $p \in P$ is the proposer, q is the sequence number and v is the value. Sequence numbers are partitioned among proposers so that no two proposers ever use the same sequence number. Proposers go through two phases: prepare and then propose. In the prepare phase the proposer asks acceptors to approve its sequence number. If an acceptor has approved sequence number q it must reject any later proposals with sequence numbers less than q. When an acceptor approves a sequence number, it must send the proposer a copy of the highest numbered proposal it has approved. Call this the "legacy" proposal. Once the proposer has received approval messages from a majority of acceptors for its sequence number it can start sending out a proposal with the same sequence number. If the proposer has received any legacy proposals, it is required to use the value from the highest numbered one as its proposal value. If no legacy proposals were received, the proposer is free to use an arbitrary proposal value. The proposer wins if it gets a majority to approve its proposal.

Suppose $m_0 = (p_0, q_0, v_0)$ has the least sequence number of any winning proposal and let X be the set of proposals with sequence numbers greater than q_0 that have been approved by at least one acceptor. X plus m_0 includes all winning proposals and possibly any number of incomplete ones. If $m_i = (p_i, q_i, v_i) \in X$, a majority of acceptors must have approved q_i, otherwise the proposal could never have been sent. Since majority sets must intersect, there must be some a which approved q_i and also approved m_0. If a had approved q_i before it approved m_0, since $q_0 < q_i$ it would have rejected m_0. So a must have approved m_0

before it approved q_i, and so a must have returned m_0 or a better proposal (which must be in X) to p_i as a legacy. It follows that m_i uses the legacy value from some proposal $m_j \in X$ or from m_0. This is true for every proposal in X. Associate each element of $m_i \in X$ with the proposal m_j which determined its proposal value. This legacy proposal must be in X or must be equal to m_0. A trace, (m_i inherits $m_j \ldots$ inherits ...) cannot repeat elements (or else a proposer would have had to re-enter the prepare phase with the same sequence number) and X is finite. So every trace must terminate with m_0. Therefore all the values of proposals in X are equal to v_0. QED.

Note that the winning proposal with the lowest sequence number can even change. Suppose there are 3 acceptors, x,y,and z, and p_1 with sequence number 1 has obtained majority approval from 2 for its sequence number and obtained approval from acceptor x for its proposal. Then p_2 with sequence number 2 gets majority acceptance for sequence number 2 and from x and y for its proposal. At this point the lowest sequence numbered approved proposal has sequence number 2. But p_1 can now move forward and get approval from z and now its proposal, with sequence number 1, has also been approved. The value does not change.

3 Paxos live

The algorithm, so far, will block if the proposer with the highest sequence number gets approval for its sequence number from a majority of acceptors and then crashes before completing the process. No other proposer can then pass the first phase. To fix this, Original Paxos allows stuck proposers to abandon their current effort, discarding all approvals and legacy proposals, increase the sequence number, and try again. As long as sequence numbers are partitioned so that proposers never reuse sequence numbers and no two proposers ever use the same number, a blocked proposer keeps increasing its sequence number until it can proceed. Unfortunately, as is noted on page 6 of the "Paxos Made Simple" paper, now we have a possible "overtaking" problem where p_1 is making progress until p_2 increases its sequence number and blocks p_1, forcing p_1 to start over with a higher number and so on forever. It is pretty easy to show this problem in a simulation.

The "livelock" problem is addressed in "Paxos Made Simple" by suggesting a single proposer be selected using timeouts! This is an odd solution because it makes most of the algorithm completely useless: there is no point in having a prepare phase and mechanisms for proposal inheritance that never happens if there is only a single proposer. It is also odd because the whole premise of Original Paxos is that the network is completely asynchronous. Once we admit that networks exist in real-time, it might be better to fix Original Paxos by imposing a staggered timeout on increasing the sequence number because failure of the highest numbered original proposal should be a rare event.

On the other hand, if we have a single proposer, then the most trivial algorithm is safe and also live (given some liveness assumption about the network, such that messages are delivered after some number of tries). The trivial algorithm just has the single proposer send its proposal until a majority of acceptors agree. There doesn't seem to be any advantage for single proposer Paxos over the trivial algorithm. The trivial algorithm can be made to survive failures of the single proposer in the obvious way, by having timeouts and election of a new proposer. If we are going to need this mechanism in Single Proposer Paxos, then using Paxos rather than the trivial algorithm would be a strange choice.

4 Next week

The next part will show how to extract the argument above from Paxos and how to use state machines to represent the algorithm.

References

[1] Leslie Lamport. "Paxos Made Simple". In: (Dec. 2001), pp. 51–58. url: `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

Author Address
`victor.yodaiken@gmail.com`