Paxos Made Simple

wu

April 19, 2024

1 The Consensus Algorithm

1.1 The problem

Assume a collection of processes that can propose values. A consensus algorithm ensures that a single one among the proposed values is chosen. If no value is proposed, then no value should be chosen. If a value has been chosen, then processes should be able to learn the chosen value. The safety requirements for consensus are:

- Only a value that has been proposed may be chosen
- Only a single value is chosen
- A process never learns that a value has been chosen unless it actually has been

The **goal** is to ensure that some proposed value is eventually chosen and, if a value has been chosen, then a process can eventually learn the value.

We let the three roles in the consensus algorithm be performed by three classes of agents: **proposers**, **acceptors** and **learners**.

Assume that agents can communicate with one another by sending messages. We use the customary asynchronous, non-Byzantine model, where:

- Agents operate at arbitrary speed, may fail by stopping, and may restart. Since all agents may fail after a value is chosen and then restart, a solution is impossible unless some information can be remembered by an agent that has failed and restarted.
- Messages can take arbitrarily long to be delivered, can be duplicated and can be lost, but they are not corrupted.

1.2 Choosing a Value

Instead of a single acceptor, let's use multiple acceptor agents. A proposer sends a proposed value to a set of acceptors. An acceptor may **accept** the proposed value. The value is chosen when a large enough set of acceptors have accepted it.

In the absence of failure or message loss, we want a value to be chosen even if only one value is proposed by a single proposer. This suggests the requirement:

P1. An acceptor must accept the first proposal that it receives.

But this requirement raises a problem. Several values could be proposed by different proposers at about the same time, leading to a situation in which every acceptor has accepted a value, but no single value is accepted by a majority of them. Even with just two proposed values, if each is accepted by about half the acceptors, failure of a single acceptor could make it impossible to learn which of the values was chosen.

P1 and the requirement that a value is chosen only when it is accepted by a majority of acceptors imply that **an acceptor must be allowed to accept more than one proposal**. We keep track of the different proposals, so a proposal consists of a proposal number and a value. To prevent confusion, we require that **different proposals have different numbers**. A value is chosen when a single proposal with that value has been accepted by a majority of the acceptors. In that case, we say that the proposal has been **chosen**.

We can allow multiple proposals to be chosen, but we must guarantee that all chosen proposals have the same value. By induction on the proposal number, it suffices to guarantee:

P2. If a proposal with value v is chosen, then every higher-numbered proposal that chosen has value v.

P2 guarantees the crucial safety property that only a single value is chosen.

To be chosen, a proposal must be accepted by at least one acceptor. So, we can satisfy P2 by satisfying

P2^a. If a proposal with value v is chosen, then every higher-numbered proposal accpeted by any acceptor has value v.

Because communication is asynchronous, a proposal could be chosen with some particular acceptor c never having received any proposal. Suppose a new proposer "wakes up" and issues a higher-numbered proposal with different value. P1 requires c to accept this proposal, violating P2^a. Maintaining both P1 and P2^a requires strengthening $P2^a$ to:

P2^b. If a proposal with value v is chosen, then every higher-numbered

proposal issued by an proposer has value v.

To discover how to satisfy P2^b, let's consider how we would prove that it holds. Assume proposal with number m has value v (H1). Now we use induction to prove that all proposal with number $n \geq m$ has value v: For $n > m$, assume all proposal with number$\in [m, n)$ has value v. For the proposal numbered m to be chosen, there must be some set C consisting of a majority of acceptors s.t. every acceptor in C accepted it. Combining this with the induction:

Proposition 1.1. *(H2): Every acceptor in C has accepted a proposal with number in $m ... (n-1)$, and every proposal with number in $m ... (n-1)$ accepted by any acceptor has value v.*

Since any set S consisting of a majority of acceptors contains at least one member of C, we can conclude that a proposal numbered n has value v by ensuring:

P2^c. For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors s.t. either

1. no acceptor in S has accepted any proposal numbered less than n
2. v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptor in S

Proposition 1.2. *P2^c + H1 + H2 $\Rightarrow$ P2^b*

Proof. Suppose a proposal with number n and value $v' \neq v$ is issued. Then there is two cases:

1. $n < m$ since there is at least one acceptor from C
2. If $n > m$, then by H2, $v' = v$.

□

To maintain the invariance of P2^c, a proposer that wants to issue a proposal numbered n must learn the highest-numbered proposal with number less than n, if any, that has been or will be accepted by each acceptor in some majority of acceptors. Instead of trying to predict the future, the proposer controls it by extracting a promise that there won't be any such acceptances. In other words, the proposer requests that the acceptors not accept any more proposals numbered less than n. This leads to the following algorithm for issuing proposals:

1. A proposer chooses a new proposal number n and sends a request to each member of some set of acceptors, asking it to respond with:
   (a) A promise never again to accept a proposal numbered less than n
   (b) The proposal with the highest number less than n that it has accpeted, if any. This is called a **prepare** request with number n.
2. If the proposer receives the requested responses from a majority of the acceptors, then it can issue a proposal with number n and value v, where v is the value of the highest-numbered proposal among the responses, or is any value selected by the proposer if the responders reported no proposals

A proposer issues a proposal by sending, to some set of acceptors, a request that the proposal be accepted. Let's call this an **accept** request.

This describes a proposer's algorithm. What about an acceptor? It can receive two kinds of requests from proposers: *prepare* requests and *accept* requests. An acceptor can ignore any request without compromising safety. So we only need to say when it is allowed to respond to a request. It can always respond to a *prepare* request. It can respond to **an** *accept* request, accepting the proposal, iff it has not promised not to. In other words:

P1^a. An acceptor can accept a proposal numbered n iff it has not responded to a *prepare* request having a number greater than n.

We now have a complete algorithm for choosing a value that satisfies the required safety properties - assuming unique proposal numbers.

Suppose an acceptor receives a *prepare* request numbered n, but it has already responded to a *prepare* request numbered greater than n, thereby promising not to accept any new proposal numbered n. There is then no reason for the acceptor to respond to the new *prepare* request, since it will not accept the proposal numbered n that the proposer wants to issue. So we have the acceptor ignore such a *prepare* request. We also have it ignore a *prepare* request for a proposal it has already accepted.

With this optimization, an acceptor only needs to remember:

1. the highest-numbered proposal that it has ever accepted
2. the number of the highest-numbered *prepare* request to which it has responded.

Because P2^c must be kept invariant regardless of failures, an acceptor must remember this information even if it fails and then restarts. Note that the

proposer can always abandon a proposal and forget all about it—as long as it never tries to issue another proposal with the same number.

The ALGORITHM:

Phase 1:

1. A proposer selects a proposal number n and sends a *prepare* request with number n to a majority of acceptors.

2. If an acceptor receives a *prepare* request with number n greater than that of any *prepare* request to which it has already responded, then it responds to the request with a promise not to accept any more proposals numbered less than n and with the highest-numbered proposal that it has accepted.

Phase 2:

1. If the proposer receives a response to its *prepare* requests (numbered n) from a majority of acceptors, then it sends an *accept* request to each of those acceptors for a proposal numbered n with a value v, where v is the value of the highest-numbered proposal among the responses, or is any value if the responses reported no proposals.

2. If an acceptor receives an *accept* request for a proposal numbered n, it accepts the proposal unless it has already responded to a *prepare* request having a number greater than n.

If an acceptor ignores a *prepare* or *accept* request because it has already received a *prepare* request with a higher number, then it should probably inform the proposer, who should then abandon its proposal.

1.3 Learning a Chosen Value

To learn that a value has been chosen, a learner must find out that a proposal has been accepted by a majority of acceptors. The obvious algorithm is to have each acceptor, whenever it accepts a proposal, respond to all learners, sending them the proposal. This allows learners to find out about a chosen value as soon as possible, but it requires each acceptor to respond to each learner - a number of responses equal to the product of the number of acceptors and the number of learners.

The assumption of non-Byzantine failures makes it easy for one learner to find out from another learner that a value has been accepted. We can have the acceptors respond with their acceptances to a distinguished learner,

which in turn informs the other learners when a value has been chosen. This approach requires an extra round for all the learners to discover the chosen value. It is also reliable, since the distinguished learner could fail. But it requires a number of responses equal only to the sum of the number of acceptors and the number of learners.

More generally, the acceptors could respond with their acceptances to some set of distinguished learners, each of which can then inform all the learners when a value has been chosen. Using a larger set of distinguished learners provides greater reliability at the cost of greater communication complexity.

Because of message loss, a value could be chosen with no learner ever finding out. The learner could ask the acceptors what proposals they have accepted, but failure of an acceptor could make it impossible to know whether or not a majority had accepted a particular proposal. In that case, learners will find out what value is chosen only when a new proposal is chosen. If a learner needs to know whether a value has been chosen, it can have a proposer issue a proposal, using the algorithm described above.

1.4 Progress

It's easy to construct a scenario in which two proposers each keep issuing a sequence of proposals with increasing numbers, none of which are ever chosen. Proposer p completes phase 1 for a proposal number n_1. Another proposer q then completes phase 1 for a proposal number $n_2 > n_1$. Proposer p's phase 2 *accept* requests for a proposal numbered n_1 are ignored numbered less than n_2. So, proposer p then begins and completes phase 1 for a new proposal number $n_3 > n_2$, causing the second phase 2 *accept* requests of proposer q to be ignored.

To guarantee progress, a distinguished proposer must be selected as the only one to try issuing proposals. If the distinguished proposer can communicate successfully with a majority of acceptors, and if it uses a proposal with number greater than any already used, then it will succeed in issuing a proposal that is accepted. By abandoning a proposal and trying again if it learns about some request with a higher proposal number, the distinguished proposer will eventually choose a high enough proposal number.

Question: Why does a distinguished proposer guarantees the progress? SE's related discussion.

If enough of the system (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer. [FLP85] implies that a reliable algorithm for

electing a proposer must use either randomness or real time - for example, by using timeouts. However, safety is ensured regardless of the success or failure of the election.

1.5 The Implementation

All that remaining is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number. Different proposers choose their numbers from disjoint sets of numbers.

2 Implementing a State Machine

To guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the ith instance being the ith state machine command in the sequence. Each server plays all the roles in each instance of the algorithm. For now, assume that the set of servers is fixed, so all instances of the consensus algorithm use the same set of agents.

In normal operation, a single server is elected to be a leader, which acts as the distinguished proposer. Clients send commands to the leader, who decides where in the sequence each command should appear. If the leader decides that a certain client command should be the 135th command, it tries to have that command chosen as the value of the 135th instance of the consensus algorithm.

Key to the efficiency of this approach is that, in the Paxos consensus algorithm, the value to be proposed is not chosen until phase 2.

New leader: When the old leader has just failed and a new leader has been selected, the new leader should know most of the commands that have already been chosen. Suppose it knows commands 1-134, 138 and 139, it then executes phase 1 of instances 135-137 and of all instances greater than 139. Suppose that the outcome of these executions determine the value to be proposed in instances 135 and 140, but leaves the proposed value unconstrained in all other instances. The leader then executes phase 2 for instances 135 and 140, thereby choosing commands 135 and 140.

Commit: The leader, as well as any other server that learns all the commands the leader knows, can now execute commands 1-135. For commands 136 and 137, we let the leader fill the gap immediately, by proposing, as commands 136 and 137, a special "no-op" command that leaves the state unchanged by phase 2 of instances 136 and 137. Once these no-op have

been chosen, commands 138-140 can be executed.

Commands 1-140 have now been chosen. The leader has also completed phase 1 for all instances greater than 140 of the consensus algorithm, and it is free to propose any value in phase 2 of those instances. It assigns command number 141 to the next command requested by a client, proposing it as the value in phase 2 of instance 141 of the consensus algorithm. It proposes the next client command it receives as command 142, and so on.

The leader can propose command 142 before it learns that its proposed command 141 has been chosen. It's possible for all the messages it sent in proposing command 141 to be lost, and for command 142 to be chosen before any other server has learned what the leader proposed as command 141. When the leader fails to receive the expected response to its phase 2 messages in instances 141, it will retransmit those messages. If all goes well, its proposed command will be chosen. However, it could fail first, leaving a gap in the sequence of chosen commands. In general, suppose a leader can get α commands ahead, a gap of up to $\alpha - 1$ commands could then arise.

A newly chosen leader executes phase 1 for infinitely many instances of the consensus algorithm. Using the same proposal number for all instances, it can do this by sending a single reasonably short message to the other servers. In phase 1, an acceptor responds with more than a simple OK only if it has already received a phase 2 message from some proposer. Thus, a server can respond for all instances with a single reasonably short message. This will be a huge performance optimization.

Since failure of the leader and election of a new one should be rare events, the effective cost of executing a state machine command—that is, of achieving consensus on the command/value—is the cost of executing only phase 2 of the consensus algorithm.

This discussion of the normal operation of the system assumes that there is always a single leader, except for a brief period between the failure of the current leader and the election of a new one. In abnormal circumstances, the leader election might fail.

- If no server is acting as leader, then no new commands will be proposed.
- If multiple servers think they are leaders, then they can propose values in the same instance of the consensus algorithm, which could prevent any value from being chosen.

  However, safety is preserved. Election of a single leader is needed only to ensure progress.

3 Thinking

Why do we need a leader in Raft? Guess the same reason as paxos.

How to implement it? There seems to be a lot of gap between the theory and the real-world implementations.

1. How to become a leader?

4 References

References

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.