Consensus algorithms for distributed systems

Märt Bakhoff, *ds.cs.ut.ee*

Abstract—Informal analysis of Raft and Paxos consensus algorithms for building practical distributed systems.

Keywords—distributed systems, consensus, paxos, raft

1 Introduction

Consensus is a problem that arises in distributed systems that are replicating a common state (such as data in a database). To keep the state consistent, each replica must apply the same operations in the same order to their copy of the state. Consensus algorithms deal with keeping the state consistent in multiple unreliable asynchronously connected replicas. This involves dealing with failures and making sure that once a value has been commited, the decision is final and not overwritten by future commits.

This paper will look at two of the more popular consensus algorithms, Raft and Paxos, and compare them in terms of capabilities and easyness to implement. For simplicity we'll assume that the cluster members are fixed.

2 Raft

Raft is a consensus algorithm that is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems.

raftconsensus.github.io

Raft works by electing a strong leader and having it coordinate a number of followers. The algorithm describes all the steps of a functional distributed system: (re)electing a leader, commiting multiple values to the transaction log and dealing with replicas failing. The cluster works as long as the majority (51%) of the nodes are online.

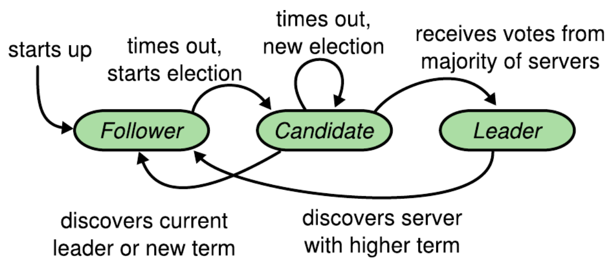
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Fig. 1. Raft server states (from [1])

2.1 Electing the leader

When a new node joins the raft cluster it starts in the *follower* state. At first the node will wait for a heartbeat message from a potential current *leader* of the cluster. If the node does not receive a heartbeat message within a time limit, it will become a leader *candidate* (Fig 1). In this case the node will send a RequestVote message to all other nodes in the cluster. Other nodes will reply with either accept or decline, based on a set of rules. If the node is accepted as a leader by a majority of the cluster, it will become the leader and start sending out heartbeat messages and commit entries to the log.

Leaders are elected for *terms*. The terms are numbered in increasing order. The leader plays an important role in Raft - only the leader is allowed to commit entries to the log. It will always know its term number and the index of the latest value commited to the log along with all the log entries.

When the leader fails, nodes will become candidates and try to replace it. There are a few rules that constrain which of the candidates will become the leader. The goal is to elect a candidate whose log contains all the entries (has the commit with the latest *commit index*).

When a candidates sends out a RequestVote message, it includes a term number (calculated from the last term known to him + 1) and his latest commit index in the message to enable other to make an informed vote.

When a follower receives a RequestVote message, it will first check if the term proposed by the sending candidate is greater than the current term known to him. If the proposed term is smaller, the vote is denied. Next the commit index is checked. If the follower has a commit index bigger than the candidate's, the vote is denied. Otherwise the follower votes for the candidate. This guarantees that there is only one leader and that leader has the latest commit index.

To be elected as leader, the candidate must receive the votes of a majority of the cluster. Also, to commit a entry to the log, the leader must successfully send the log entry to the majority of the cluster. If a candidate with an out of date commit index sends the RequestVote message, it can never get the majority of votes - if he did, there be at least one node that was in the majority that accepted a newer commit but voted for the candidate with stale commit index. That node would violate the rules for electing a leader.

There are a two main cases when several nodes can claim to be the leader. First, if the current leader fails, a new one is elected but then the old leader recovers. This is resolved quickly - when the old leader receives a heartbeat message from the new leader (with a term number bigger than his), it will drop to follower state immediately. The other case is then two candidates start an election at the same time. In that case, only one of them can get the majority of the votes. The winner will start sending heartbeat messages and other candidates will switch to follower state.

2.2 Commiting a value

All modifications to the log are coordinated by the leader in Raft. When the leader is elected, it is guaranteed that he has the most recent commited value in his log. The leader can never delete commited log entries from the log. The log in append-only and only the leader can append entries to the log.

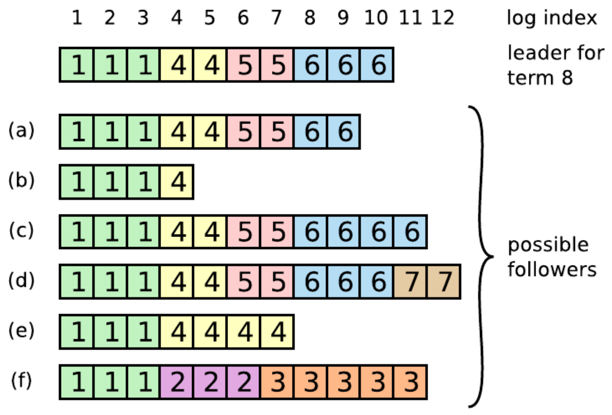

Fig. 2. Different states of log (from [1])

To commit something to the log, a client must send a request to the leader. If the client connects to a follower node, he is simply redirected to the current leader. The leader then tries to commit the entry by adding the entry to its log and sending AppendEntries message to its followers. The entry is considered commited when at least the majority of the nodes have confirmed the AppendEntries command.

All entries in the log are numbered. If a failed node recovers and rejoins the cluster as a follower, it will have some of the log entries missing. In this case the next AppendEntries command to it will fail, because the commit indexes are not sequential. The follower will reject the AppendEntries command and send his current commit index in the reply. The leader will then help the follower to "catch up" by sending it all the log entries since it failed (determined by the followers commit index).

The follower's log can be longer than the leader's (Fig 2). That happens if the follower is a former leader who crashed before commiting some of the entries (he added the entries to his own log but didn't get the majority's support). In this case the follower simply deletes the extra entries from it's log, since they cannot have been commited anyway.

2.3 Handling failures

The safety properties of Raft rely on using the majority of nodes to make decisions. Because of that, the algorithm can only work if the majority of the nodes is online. If half the nodes or more fail, the algorithm will simply stop making progress - new leaders cannot be elected and no new entries can be considered commited. Otherwise the failure of up to $n/2-1$ nodes will not affect the availability of the cluster.

If the leader fails the algorithm must elect a new leader and make sure the commited entries are not lost. The election (as described above) will guarantee that the new leader's log will contain the most recent commited entry. When elected, the leader will replicate his log to all the followers.

When a leader crashes, then recovers and sends AppendEntries to other nodes before he receives the new leaders heartbeat, the other nodes will simply ignore his commands. The old leader is detected by the stale term number in his AppendEntries message.

3 PAXOS

Paxos is a considerably older algorithm than Raft. It was first described in 1990 by L. Lamport and was the first consensus algorithm that had a formal proof of correctness. The algorithm was allegedly born out of an attempt to prove that such algorithm was impossible to create. Due to it's origin, the original papers on Paxos are mathematically formal and focus more on the properties and proofs of the algorithm rather than the desciription of a practical implementation.

3.1 Overview

Basic Paxos is an algorithm for deciding on a single value by a cluster of nodes. Deciding on multiple values is an extension to the algorithm and described in less detail by many of the materials about Paxos.

There are three types of nodes (processes) in a Paxos system: *proposers, accepters and learners* (Fig 3). Proposers propose values that should be chosen by the consensus. Acceptors form the consensus and accept values. The acceptors' decisions are local and no single acceptor knows the decision of the majority. Learners are a valueable source of information because they learn which value was chosen by each acceptor and therefore the consensus. A process may fill several roles at a same time. For example, a database client is likely to be a proposer and also a learner.

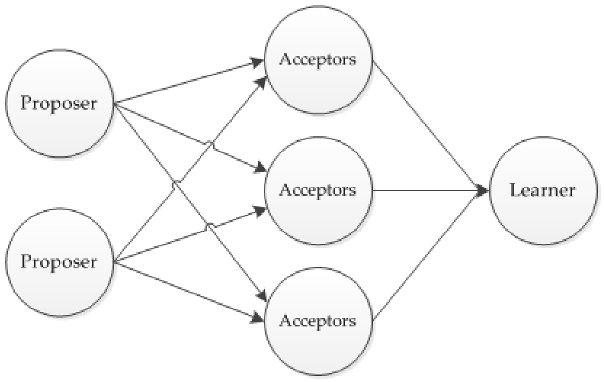

Fig. 3. Processes in Paxos (from [4])

3.2 Making a proposal

The objective of Paxos is to choose a *single value*, even though several proposers may propose different values in different order. For a value to be considered chosen, the majority of the acceptors must inform a learner about accepting that value.

The process of an acceptor accepting a value has two phases - *prepare* and *accept*. A proposer starts the protocol by first gathering information from the acceptors. He sends a 'prepare' message with a *proposal number n* to the acceptors. The proposal numbers form a timeline in which the proposal with the biggest number is considered up-to-date and proposals with smaller numbers signify out-of-date information.

When an acceptor receives a prepare message, it will compare the proposal number with the biggest proposal number it has seen so far. If the acceptor has seen a proposal with a bigger number, then the proposer is missing the latest information in the proposals timeline and the prepare message is rejected. If the proposal number is bigger than any number seen so far then the acceptor recognizes that proposer has up-to-date information about the timeline. The proposal number then becomes the biggest proposal number that the acceptor has seen and future attempts to propose values with a smaller proposal number are rejected.

In any case, the acceptor responds to the prepare message. The response includes:

1) whether the prepare was accepted,
2) the biggest number the acceptor has seen,
3) if the acceptor has already accepted any values, then the accepted value is also included.

The proposer must receive a response from the majority before proceeding.

3.3 Accepting a value

The proposer will receive a response from the majority (or more) of the acceptors. If the majority of the acceptors rejected the prepare, then the proposer should update his information about the timeline of proposals (received from the acceptors) and start over.

If the majority of the acceptors accepted the prepare, the proposer has hope to finish the protocol successfully. The proposer should analyse the responses from the accepters. If the majority of the accepters have already accepted a value (included in their response) then the decision has already been made about the value and the original value of the proposer cannot be chosen in this Paxos instance (remember that Paxos is for deciding on only a single value).

If the majority of the acceptors have not yet accepted a value then the proposer can proceed with having the acceptors accept the value proposed by him. The proposer will send the acceptors an 'accept!' message along with his proposal number n and the value that should be accepted by the acceptors. If everything goes well, the acceptors will mark the value as accepted and notify the learners about their decision.

3.4 Handling failures

Paxos can only work if the majority of the acceptors are online. If more than $n/2 - 1$ acceptors fail, no proposer can get a reply to his prepare messages from the majority and no values can be accepted.

Another weak point of Paxos is the timing of messages sent by proposers. In case of a single proposer, the single proposer will always have the proposal with the biggest number and no proposals are rejected. In case of multiple proposers, the proposers can begin to interfere with each other.

Consider the case when a proposer has received confirmations to his prepare message from the majority but has yet to send the accept! messages. Another proposer can send a competing prepare message to the accepters with a higher proposal number and therefore block the first proposer's proposal from being accepted.

This process, called "dueling proposers", can go on in cycles infinitely. The Paxos algorithm does not give a single solution to it and claims

> "more non-determinism is better, because it allows more implementations."

In practice, the problem is usually solved by choosing a weak leader using some fault detection algorithm or simply using random exponential backoff (as used in ethernet).

4 Practical comparison

4.1 Capabilities

Provably correct. Both algoritms have proofs of correctness. Lamport's Paxos has a rigorous mathematical proof for many aspects of the algorithm. Raft has less formal proofs but all aspects are well explained and main aspects are proven.

Fit for purpose. Both algorithms successfully solve the problem of deciding a value within a set of unreliable asynchronous nodes. However, Basic Paxos only solves the problem for deciding a single value while making multiple decisions is an integral part of the Raft protocol.

4.2 Properties

Both algorithms provide validity (only values that are proposed can be chosen) and agreement (if a value is chosen, then no other value can be chosen) but not termination. When the majority of the nodes are offline, the algorithms simply stop making progress.

4.3 Implementations

Even though Raft and Paxos solve more or less the same problem, the goals of their authors are radically different.

Paxos' author focuses on thoroughly proving the algorithm and providing a framework for building your own consensus algorithm on top of it. Raft on the other hand focuses on describing a complete algorithm that is ready to be implemented without having to invent anything to make the end result useful.

The difference is also visible in the implementations of the algorithms. There are may implementations of both of them, but there seem to be more (public) Raft implementations and they are a lot more uniform. There's a list of nearly 50 open source implementations of Raft on it's official homepage at raftconsensus.github.io.

Paxos also has a list of publicly known implementations (wikipedia lists over 10 implementations). However it seems that impementing Paxos is a lot more difficult because the authors of Paxos did not seem to consider practical implementations when designing the algorithm:

> Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons
>
> Google Chubby team [5]

5 Conclusion

Both Paxos and Raft manage to solve the consensus problem using the majorities of the cluster. They differ mostly by their focus (Raft aims to provide a complete practical algorithm whereas Paxos provides the building blocks of a consensus algorithm) and features (single decision vs multiple decisions, strong leader vs weak leader).

Acknowledgments

The author would like to thank Oleg Batrašev, Taivo Käsper and the distributed systems group at University of Tartu.

References

[1] Diego Ongaro and John Ousterhout, *In Search of an Understandable Consensus Algorithm*, Stanford University, February 22, 2014

[2] Leslie Lamport, *Paxos Made Simple*, ACM SIGACT News 32, December 2001

[3] James Aspnes, *Notes on Theory of Distributed Systems*, Yale University, 2014

[4] Angus Macdonald, *Distributed Consensus: Paxos by Example*, May 2013

[5] Tushar Chandra et.al., Paxos Made Live - An Engineering Perspective, ACM PODC '07