Three modifications for the Raft consensus algorithm

Henrik Ingo
henrik.ingo@openlife.cc
August 2015
(v0.2, fixed bugs in algorithm, in Figure 1)

1. Background

In my work at MongoDB, I've been involved in a team that is adapting our replication protocol to conform to the principles set out in the academic paper that describes the Raft algorithm(Ongaro, 2014). Breaking with the academia's long standing obsession with Paxos, of which I'm yet to hear about a robust real world and open source implementation, Raft describes a simple, easy to understand leader based consensus algorithm. (In the vocabulary used outside of academia, Raft describes a single-master, synchronous replication protocol using majority elections.)

Hence, while my original interest in Raft is purely work related, clearly it is meaningful to consider Raft itself as a valuable contribution to the brittle art of making databases highly available. As such, this paper is written purely within the context of Raft itself, without any reference to the replication implementation in MongoDB, and ignoring the multitude of differences that a MongoDB implementation will have compared to Raft. (One of which is MongoDB being pull based, while Raft is a push based algorithm.)

2. Introduction

The major contribution of the Raft authors has clearly been to finally introduce an easy to understand, easy to implement protocol for distributed consensus. While more complicated protocols may offer additional value in terms of feature set - such as supporting a multi-master scheme - it is a valuable resource for the community of distributed databases to at least have a robust alternative for the most basic kind of replication, leader based consensus. Somewhat surprisingly, the level of elegance achieved by Raft has not been clearly documented previously, let alone provided with proofs of correctness. (The closest, and compared to Paxos rather useful, relative to Raft would be the family of ViewStamped replication algorithms(Oki & Liskov, 1988), however Raft significantly simplifies compared to ViewStamped replication.)

It is common for academic papers to focus mostly on the core replication of events themselves, and the closely related mechanism of electing a leader or determining a quorum. On the other hand they commonly neglect important surrounding topics, such as cluster maintenance activities or "environmental corner cases", that in the real world are equally important ingredients in creating a complete solution for a highly available database.

Also the Raft algorithm has evolved in similar fashion. This is evident when following the succession of published papers from the original Usenix 2014 paper[(Ongaro & Ousterhout, Usenix 2014)] to the extended version of the same paper[(Ongaro & Ousterhout, 2014)] to the thesis by Diego Ongaro published later in 2014[(Ongaro, 2014)].

For example, the original Usenix paper did not include any mention of snapshotting the state machine, rather simply describes an indefinitely growing log of events - clearly not realistic for most real world systems that might implement Raft, including the authors' own Ramcloud database. The extended version then added a discussion on snapshotting, including an InstallSnapshot RPC for sending the snapshot to followers when needed.

Similarly the original Usenix paper does include a discussion on cluster membership changes, but it is obvious even to the casual reader that this part of the paper did not receive the same amount of thought that went into the core of the algorithm, and certainly does not achieve the simplicity goal the authors had set themselves. Ultimately the cluster membership change protocol ends up in the curious state where members of the cluster are receiving (and accepting!) events from a leader that's not even part of the current cluster. The Ongaro thesis then replaces that complexity with 2 very simple RPCs to add and remove servers one at a time. And as is often the case, the simpler algorithm also turns out to be more robust than the complex one!

In the same spirit of evolving from a core protocol to a more complete and realistic implementation, the goal of this paper is to introduce 3 modifications to Raft, that are relevant to real-world distributed databases:

1. Universally Unique Database Identifier: to identify whether a snapshot or a log on one server is in fact some (predecessor or successor) state of the same state machine on other servers of the cluster, or whether it's a snapshot of some completely different state machine that has nothing to do with this cluster.
2. Pre-vote algorithm: this paper provides a more detailed specification of the idea suggested only in passing in §9.6 in (Ongaro, 2014)
3. Leader stickiness: Building on the pre-vote algorithm, we also modify Raft to reject servers that attempt to elect themselves as leader, if the current leader appears to still be healthy to the rest of the cluster. This is to avoid flip-flopping between two competing leaders.

The proposed modifications in this paper are written against the most recent publication of Raft, Diego Ongaro's thesis paper[(Ongaro, 2014)], which the reader is assumed to be familiar with. The tables summarizing the algorithm have been reproduced on the next few pages. The additions of this paper are highlighted in dark blue.

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

databaseId	unique, constant identifier generated at first boot to initialize the empty database. Note: a forced reconfiguration of the cluster also counts as first boot (of a new cluster)
currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on all leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§3.5), also used as heartbeat (§3.4).

Arguments:

databaseId	The unique identifier for this database
term	leaders term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. If databaseId of caller doesn't match: delete all snapshots and log and reset to initial state, set databaseId from caller.
2. Reply false if term < currentTerm (§3.3)

RequestVote RPC

Invoked by candidates to gather votes (§3.4).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§3.6)
lastLogTerm	term of candidate's last log entry (§3.6)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if last AppendEntries call was received less than election timeout ago
2. Reply false if term < currentTerm (§3.3)
3. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

Rules for Servers

All servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§3.5)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§3.3)

Followers (§3.4):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§3.4)

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election time elapses: start new election

3. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
4. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
5. Append any new entries not already in the log
6. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Leaders:

- Upon election: send initial empty AppendEntries RPC (heartbeat) to each server, repeat during idle periods to prevent election timeouts (§3.4)
- If command received from client: append entry to local log, respond after entry applied to state machine (§3.5)
- If last log index >= nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§3.5)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] >= N, and log[N].term == currentTerm: set commitIndex = N (§3.5, §3.6)

AddServer RPC

Invoked by admin to add a server to the cluster configuration.

Arguments:

newServer	address of server to add to configuration

Results:

status	OK if server was added successfully
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader (§6.2)
2. Optional: If leader.databaseId != newServer.databaseId, and new server has a non-empty state, reply with error and instruct admin to reset the state first (e.g. delete database).
3. Catch up new server for fixed number of rounds. Reply TIMEOUT if new server does not make progress for an election timeout or if the last round takes longer than the election timeout. (§4.2.1)
4. Wait until previous configuration in log is committed (§4.1)
5. Append new configuration entry to log (old configuration plus newServer), commit it using majority of new configuration (§4.1)
6. Reply OK

RemoveServer RPC

Invoked by admin to remove a server from the cluster configuration.

Arguments:

oldServer	address of server to remove from configuration

Results:

status	OK if server was removed successfully
leaderHint	address of recent leader, if known

Receiver implementation:

1. Reply NOT_LEADER if not leader (§6.2)
2. Wait until previous configuration in log is committed (§4.1)
3. Append new configuration entry to log (old configuration without oldServer), commit it using majority of new configuration (§4.1)
4. Reply OK and, if this server was removed, step down (§4.2.2)

Raft State for Compaction

Persisted before discarding log entries. Also sent from leader to slow followers when transmitting state.

prevIndex	index of last discarded entry (initialized to 0 on first boot)
prevTerm	term of last discarded entry (initialized to 0 on first boot)
databaseId	the unique identifier for this database
prevConfig	latest cluster membership configuration up through prevIndex

InstallSnapshot RPC

Invoked by leader to send chunks of a snapshot to a follower. Leaders always send chunks in order.

Arguments:

databaseId	the unique identifier for this database
term	leader's term
leaderId	so follower can redirect clients
lastIndex	the snapshot replaces all entries up to and including this index
lastTerm	term of lastIndex
lastConfig	latest cluster configuration as of lastIndex

PreVote RPC (NEW)

Called by a server before changing itself to Candidate status. If a majority of servers return true, proceed to Candidate. Otherwise, wait for another election timeout.

Arguments:

nextTerm	caller's term + 1
candidateId	caller
lastLogIndex	index of caller's last log entry
lastLogTerm	term of caller's last log entry

Results:

term	currentTerm, for caller to update itself
voteGranted	true means caller would receive vote if it was a candidate

Receiver implementation:

1. Reply false if not in candidate state (leader stickiness)
2. Reply false if nextTerm < currentTerm
3. If caller's log is is at least as up-to-date as receiver's log, return true

	(include only with first chunk)
offset	byte offset where chunk is positioned in the snapshot file
data[]	raw bytes of the snapshot file, starting at chunk
done	true if this is the last chunk

Results:

term	currentTerm, for leader to update itself

Receiver implementation:

1. If databaseId of caller doesn't match: delete all snapshots and log and reset to initial state. Set databaseId to the new value given in arguments.
2. Reply immediately if term < currentTerm
3. Create new snapshot file if first chunk (offset is 0)
4. Write data into snapshot file at given offset
5. Reply and wait for more data chunks if done is false
6. If lastIndex is larger than latest snapshot's, save snapshot file and Raft state (lastIndex, lastTerm, lastConfig). Discard any existing or partial snapshot.
7. If existing log entry has same index and term as lastIndex and lastTerm, discard log up to through lastIndex (but retain any following entries) and reply
8. Discard the entire log
9. Reset state machine using snapshot contents (and load lastConfig as cluster configuration)

Figure 1: Summary of the Raft algorithm, from (Ongaro, 2014). References to a section (like §3.4) reference sections in (Ongaro, 2014). Additions proposed in this paper are highlighted in dark blue. The PreVote RPC is new in its entirety, though the idea was suggested in (Ongaro, 2014).

3. Universally Unique Database Identifier

While the Raft algorithm as published in (Ongaro, 2014) does a great job in maintaining the integrity of a single replicated state machine, in the real world database clusters don't live in a vacuum. A sysadmin will be operating multiple servers in one or more datacenters, each server belonging to some cluster. Failing to take this obvious fact into account, implementations will often be left vulnerable to various split brain conditions, especially due to operational errors such as misconfiguration. While we could blame many of these conditions on the sysadmin, it is actually simple for an implementation to protect against such errors, and one should of course therefore do so.

To illustrate a simple path to a split brain, one out of many possible, consider a 2 server cluster and the following steps, that are fully legitimate operations as defined in the (Ongaro, 2014) version of Raft:

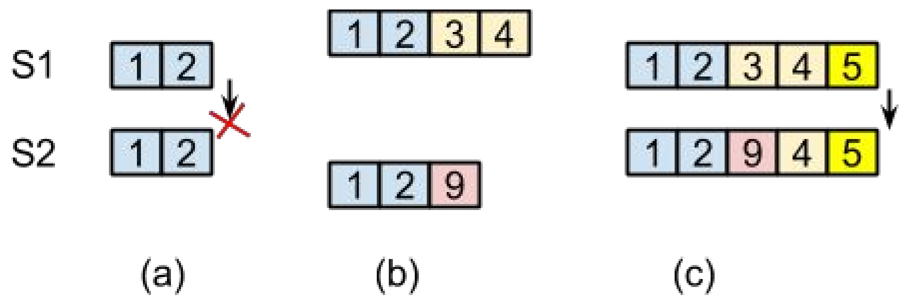

Figure 2: *Sequence of steps leading to split brain, which goes undetected:*
a) Two servers S1 and S2 form the cluster, with S1 being the leader. They commit x<-1 and y<-2, after which there is a network split. Since neither server alone is a majority, there is no leader and replication stops. At this point both servers have (term=1, index=2).
b) To get the service back into available state, the sysadmin has to change the cluster configuration by calling S1.RemoveServer(S2). (Strictly speaking this can only be called on the leader, however since it is not possible for any server to become leader in this state, a real world implementation would provide - or the sysadmin would in any case invent - some method of forcing the reconfiguration to get back to a functional state.) Unfortunately, another sysadmin at the same time calls S2.RemoveServer(S1), causing split brain.
S1 commits z<-3 and x<-4. (term=2, index=2)
S2 commits z<-9. (term=2, index=1)
c) As the network is restored, the sysadmins take steps to rejoin the servers with each other, calling S1.AddServer(S2). S1 then calls AppendEntries RPC to compare the term and index of each of the servers logs, and sending the one log entry (x<-4) that appears to be missing, causing S2 to "catch up".
Finally, S1 commits, and replicates to S2: y<-5. (term=3, index=1)
The diverging value at (term=2, index=1) goes undetected, perpetuating a split brain between the state machines, until overwritten by a new value.

More generally, the various conditions that the AddServer RPC must be capable of dealing with, are the following:

	Event	Correct action
#1	A new server, with an empty database, is added to the cluster.	InstallSnapshot RPC

#2	A server that has previously been removed from the cluster, is added back. The snapshot files and log are in the state as they were when removed.	AppendEntries RPC If too much time has passed, the leader's log will not reach far back enough in time. In this case the new server is considered a slow follower and InstallSnapshot RPC is implicitly called.
#3	A server that was never part of this cluster, but has a non-empty database (and has currentTerm and index set to non-zero values) that was previously created by some other application, is added to the cluster.	Delete old database and reset state. InstallSnapshot RPC (Alternatively, a conservative implementation would simply fail the request, leaving it to a sysadmin to delete and reset the database.)
#4	A server that was never part of this cluster, but has been provisioned with a backup copy of the correct database (in order to speed up the process of adding it to the cluster), is added to the cluster.	AppendEntries RPC

Figure 3: Table of different kinds of scenarios that may be encountered by AddServer RPC.

The problem for the implementor of a replicated state machine is that it is not easy to distinguish between the different scenarios 2-4. A simple robust solution would be to always delete data and reset state on the added server, so that one would always commence with InstallSnapshot RPC. However, this would defeat the sysadmin's attempt at optimizing the process in scenario #4, and may often be sub-optimal also for scenario #2.

To be able to distinguish scenario #3 from the others, we will have to introduce a *Universally Unique Database Identifier*, which we will call *databaseId* in the above summary tables of the Raft algorithm.[1]

Any pseudo-unique identifier is fine, such as using a UUID. Implementations should not allow users to provide a unique identifier. That would just lead to every other databaseId having the value "mycluster".

The databaseId is generated at the creation of a new cluster and is thereafter constant and the same value for all servers in the cluster. It should be persisted to durable storage, and should typically be part of backups. Note that the triplet of (databaseId, term, index) will uniquely and globally identify a state of the database or state machine. If two state machines both are in the state (databaseId=x, term=y, index=z), they are guaranteed to be identical. Note how this extends the Log Matching property into a world where there exists more than one database cluster, making it a globally valid property.

With the method presented in (Ongaro, 2014) of bootstrapping a cluster via repeated calls to AddServer one by one, this means that the databaseId is generated by every server during their

[1] Note that the reference implementation of Raft, LogCabin (https://github.com/logcabin/logcabin) includes a global variable called clusterUUID. It does not appear to have the functionality proposed here, rather is used as part of the client protocol. Also it is set in the user configuration, and not persisted to the log or snapshot. Hence, the LogCabin UUID is, like it name suggests, used to identify the cluster of servers. Otoh, this paper is proposing a UUID to identify the state machine data(base) as data, irrespective of any server instances the data may reside on.

first boot. Servers that are then added to another cluster with AddServer RPC, will then discard their databaseId and instead adopt the one shared by the cluster they are added to.

4. Pre-vote algorithm

Section 9.6 in (Ongaro, 2014) introduced the idea of a pre-vote algorithm, without actually spelling out the details of such an algorithm. We will define one such algorithm here, so that we can build upon it in the next section.

The usefulness of a pre-vote algorithm is easily explained. The Raft algorithm has a strong property that leads its participants to always adopt the highest term they have observed from another server. This property is key to the robustness of the algorithm: elections become deterministic because of this, and the Log Matching property likewise depends on this.

A drawback in real-world implementations is that this easily leads to unnecessary "term inflation". Assuming servers will use 64-bit integer mathematics, they are unlikely to run out of numbers during the lifetime of a cluster, but clusters do become subject to a behavior where the malfunctioning server will force an election when re-joining a cluster, even if the rest of the cluster has been healthy and continues to have a fully functioning leader.

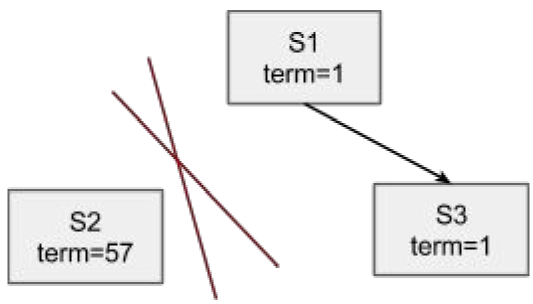

Figure 4: 0) A Raft cluster with 3 servers, one of which (S2) is temporarily disconnected from the rest.
1) Raft causes the disconnected server to call for an election once every election timeout, causing it to increment currentTerm. As it cannot connect to other servers, it will lose its election, and keep retrying.
2) Once it is able to reconnect with the rest of the cluster, its higher term will propagate to S1 and S3. This will cause S3 to stop accepting new log entries from S1, and will cause S1 to step down as leader.
3) A new election for term=58 is eventually triggered and will be won by whichever server goes ahead first.

The solution is to introduce a pre-vote algorithm, which is executed before changing to candidate status. Only if the pre-vote is successful, should the server switch to candidate, otherwise it should wait for another election timeout.

The implementation of the pre-vote algorithm is straightforward: receivers of a PreVote RPC should respond with the same result that they would if this was an actual vote.

However, it is important to emphasize that the pre-vote response is not binding on a server. While each server must only vote for a single candidate for a given term, this is not true for the pre-vote phase. Multiple to-be-candidates could receive a positive indication in the pre-vote phase, however once they proceed to the actual election, only one of them would receive the actual vote. This behavior is key to avoiding race conditions that could lead to inconclusive elections.

For example, a server could succeed in getting a prospective majority in the pre-vote phase, but then become itself disconnected before it is able to proceed with the actual election. In this case it would be a waste of precious failover time not to vote for another candidate who still can win the election.

5. Leader stickiness

The last flaw to address in this paper is Raft's vulnerability, in some corner cases, to leader flip-flopping. An example of such a scenario is shown in the following picture:

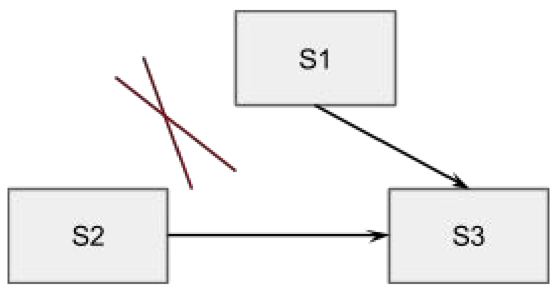

Figure 5: 0) In the starting state, S1 is a leader and S2 and S3 followers. At this specific moment, clients are not sending any new events, so nothing is added to the log on S1, but it is sending empty AppendEntries RPC calls for heartbeat purposes.
Note that the below scenario only happens when all logs in the cluster remain equal. Raft would in fact protect against flip-flopping in a case where S1 is able to successfully replicate new events to S3, since that would cause S2 to lose elections until it can catch up to the same position in its own log.
1) Network connectivity between S1 and S2 breaks down, causing each of them to conclude that the other server is disconnected from the cluster. However, both of them can still connect to S3.
2) As the election timeout is triggered on S2, it will increase its currentTerm and call RequestVote RPC. S3 will grant the vote, and S2 will become the new leader.
3) S1 will learn from S3 about the increased term, and step down as leader. However, it will not get any connection from S2, which will cause election timeout to trigger. S1 will then increase its currentTerm and call for an election. After the second attempt, S3 will grant the vote, making S1 leader again.

4) This flip-flopping continues forever until the network connectivity between S1 and S2 is restored.

It is important to understand that, on a superficial level, flip-flopping is not a failure of the Raft protocol. Superficially, all the requirements are met at all times: There is exactly one leader at any time, hence the cluster remains available. Also the Log Matching property remains intact, the integrity of data is not corrupted.

In practice however, such flip-flop behavior is undesired. It would be unwanted overhead for the clients to have to reconnect to a new leader every few seconds. For some types of implementations - such as a database supporting long lived, multi-statement transactions - it could make the cluster de-facto unavailable, if the leader's elected term is shorter than the time it takes to commit a transaction.

So instead of such flip-flops, we would expect a (healthy) cluster to keep the same leader for days and weeks, maybe even months.

Raft actually does protect against some cases of flip-flopping. Since it is a requirement for the winning server to have its log at least as up to date as a majority of the cluster, this means that a single server, or minority of servers, cannot disrupt the majority once they have fallen behind. Which they are of course likely to do as soon as they become disconnected from the leader.

The example described above therefore requires that no new log entries are committed, or at least not replicated, so it is admittedly a corner case. However, in the real world flip-flopping may often also be caused by flaky networks, where connectivity breaks down for some seconds, then is restored, then breaks down again. Such network conditions would increase the likelihood of flip-flopping compared to our simple example, since the intermittently-disconnected servers will be able to catch up with the leader. In such conditions the desired behavior would be for the cluster to keep its current leader as long as possible, rather than descending into a cycle of frequent elections and new leaders. It is with these real-world situation in mind that the following improvement is proposed.

A simple fix to such network flakiness would be to simply increase the election timeout to be longer than the typical network interruption. This would allow short disconnections to heal themselves before servers in the cluster start calling for election. And this is in fact a common and effective fix to flakiness. But increasing the election time also has the effect of increasing the failover time in every case. Hence this proposal to add "leader stickiness" to Raft can be seen as a more advanced solution. While it adds some complexity, it is motivated by minimizing failover time, or maximizing availability.

The change to add "leader stickiness" is intuitive: Followers should reject new leaders, if from their point of view the existing leader is still functioning correctly - meaning that they have received an AppendEntries call less than an election timeout ago. This will have the effect that a single "problematic" server (or even a minority of servers) with connectivity issues will not be able to win new elections, if a majority of the cluster is still connected to the current leader and functioning normally. Or, put in more political wording: followers will be loyal to their existing leader, and only servers that agree that an election is needed (Candidates) will vote for a leader from within themselves.

The intuitive fix does introduce a non-obvious change to how new leaders are chosen also in the common case. Now, with the stickiness added to the elections, the first servers to have their election timeout lapse after a leader failure, will actually be rejected in the PreVote phase, since a majority of servers will still think that the leader is there. Only when at least half of the servers have reached the election timeout, and thus converted to the Candidate state, will a server be able to gain majority support from the PreVote phase, and then ultimately from the RequestVote itself. In the simple case, for a cluster with 3 members, the second server to call PreVote RPC will become the new leader, in a 5 member cluster the third server, and so on.

References

Oki & Liskov, 1998 Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems
Brian M. Oki & Barbara H. Liskov
1988 ACM O-89791-277-2/88/0007/0008
http://pmg.csail.mit.edu/papers/vr.pdf

Ongaro, 2014: Consensus: Bridging Theory and Practice
Diego Ongaro, August 2014.
https://ramcloud.stanford.edu/~ongaro/thesis.pdf

Ongaro & Ousterhout, 2014:
In Search of an Understandable Consensus Algorithm (Extended Version)
Diego Ongaro & John Ousterhout; May 20, 2014.
https://ramcloud.stanford.edu/wiki/download/attachments/6586375/raft.pdf

Ongaro & Ousterhout, Usenix 2014:
In search of an understandable consensus algorithm.
Ongaro, D & Ousterhout, J.
In Proc ATC'14, USENIX Annual Technical Conference (2014), USENIX.