IN SEARCH OF AN UNDERSTANDABLE CONSENSUS ALGORITHM

Patroclos Christou
chrispat@di.uoa.gr
M1392

Introduction

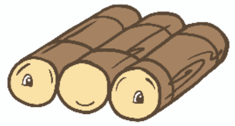
What is a Consensus algorithm

Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final.
Consensus algorithms allow a collection of machines (replicated state machines) to work as a coherent group that can survive the failures of some of its members and keep a replicated log consistent.

Properties

- Safety
- Full Functionality
- Not Time depending the Log consistency
- Server performance does not impact overall System performance

Our Algorithm - Raft



Raft is a consensus algorithm for managing a replicated log. It is efficient as Paxos and produces a result equivalent to Paxos.
Its structure is different from Paxos and this makes Raft more understandable and also with a better foundation for building practical systems.

<u>**New features on Raft**</u> *(compared with other consensus algorithms)*

- Strong leader (The flow starting from leader to the followers)
- Leader election (Based on random timers)
- Membership changes (mechanism for changing the set of servers when configuration overlaps)

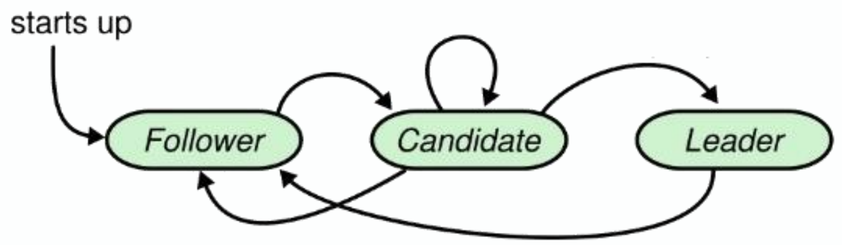
To increase Understandability, Raft decomposes the consensus problem into three relatively independent sub problems:

- **Leader election**
- **Log replication**
- **Safety**

Raft Basics

Suppose that we have a cluster with several nodes. At any given time, each of the nodes is in the one of the following states:

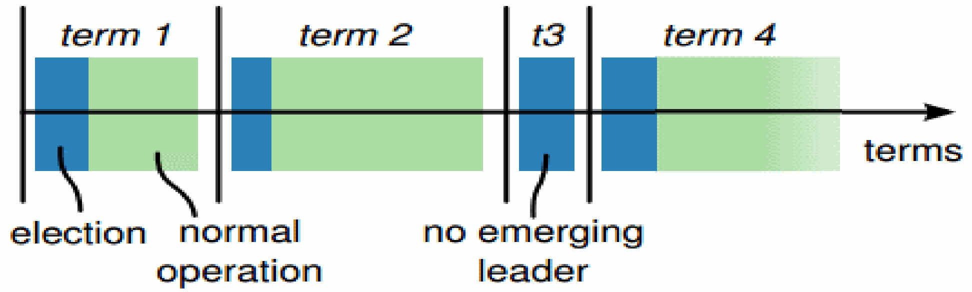
- Candidate
- Leader
- Follower



The **candidate** is used to elect a new leader.
There is exactly one **leader** and all of the other servers are followers. The leader handles all client requests,
Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates.

Raft Basics (cont.)

Raft divides time into ***terms*** of arbitrary length.
Terms are numbered with consecutive integers. Each term begins with an *election*, in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the term.
In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

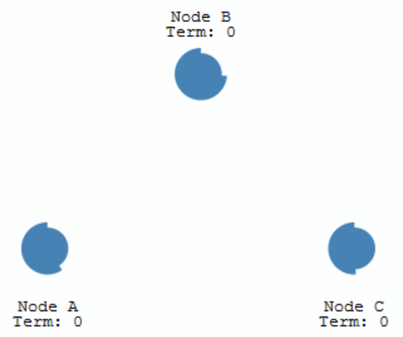


Raft servers communicate using remote procedure calls (RPCs), and the basic consensus algorithm requires only two(three) types of RPCs.

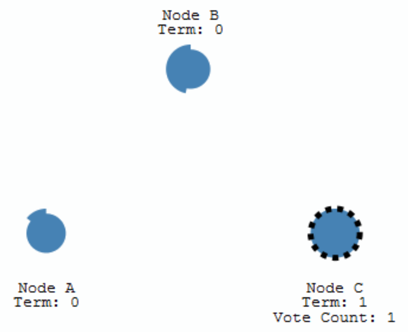
- **RequestVote** (used during election)
- **Append-Entries** (used by leaders to replicate log)
- **InstallSnapshot** (snapshot transferring between nodes)

Quick execution example

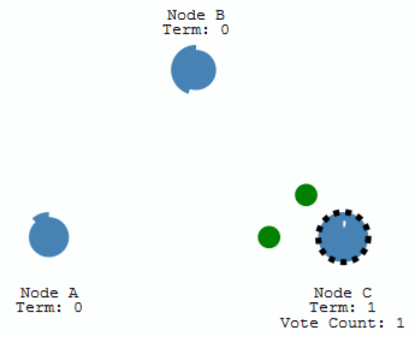
Now let’s see what happened when Raft algorithm starts between 3 nodes



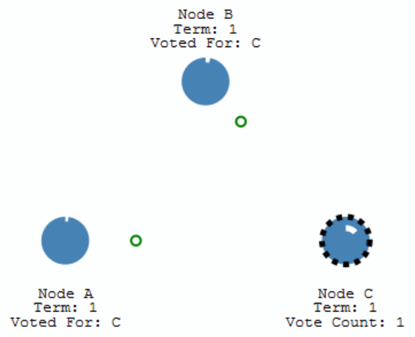
The 3 nodes starting with a random term



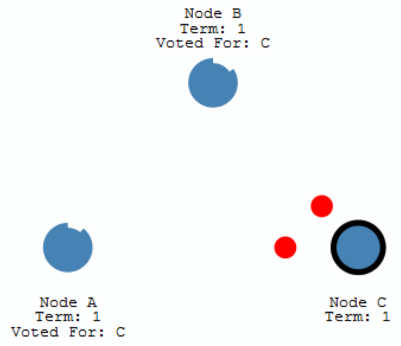
Suppose C ended first



Send RequestVote RPC to the other nodes



A,B reload their terms and response with the accepted vote



From now C is the leader. Send periodically Heatbeats(RPCs) to check remaining nodes and reload their terms

Leader Election (let's see some details)

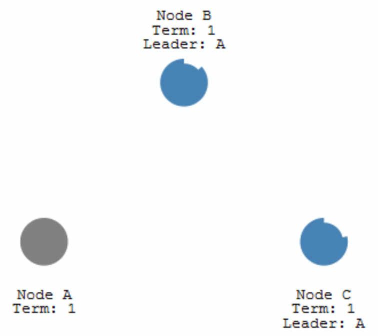
Basic Raft's mechanism is **Heartbeat**. Leader send periodic heartbeats (AppendEntriesRPCs that carry no log entries) to all followers in order to maintain their authority.

If no leader exists in cluster or someway he stopped then the following happen:

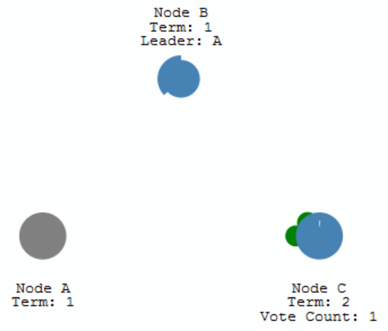
1. When follower receives no communication over a period of time (*election timeout)*, then it assumes there is no viable leader and begins an election
2. The follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other nodes in the cluster:
   A. If a candidate wins an election if it receives votes from a majority of the nodes (each server will vote for at most one candidate in a given term). **Then he becomes leader** and then sends heartbeat messages to all of the nodes to establish its authority and prevent new elections.
   B. If during election may receive an RPC from another candidate to become leader. If the other candidate's term is larger than the current's term, then the **candidate recognizes the leader as legitimate and returns to follower state. Otherwise rejected it.**
   C. If many followers **become candidates at the same time**, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term(150-300ms) and **initiating another election round.**

Execution example (leader stops)

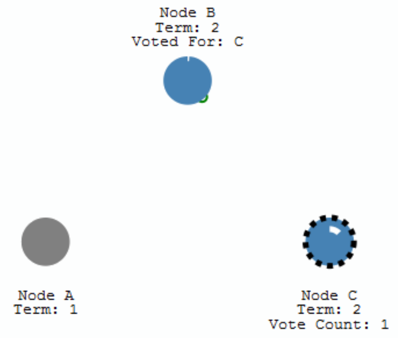
Suppose that our leader was A node



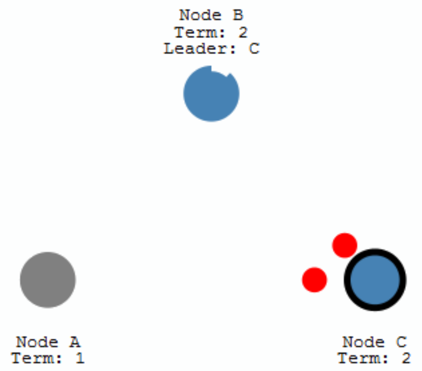
Suddenly A node(leader) stops responding

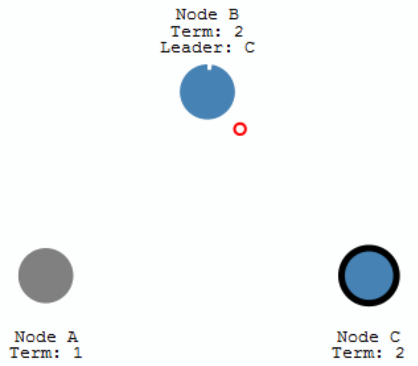


From now on, nodes does not get a Heartbeat. C’s term finish first and start as Candidate



C asking for vote to the reaming nodes. B answer





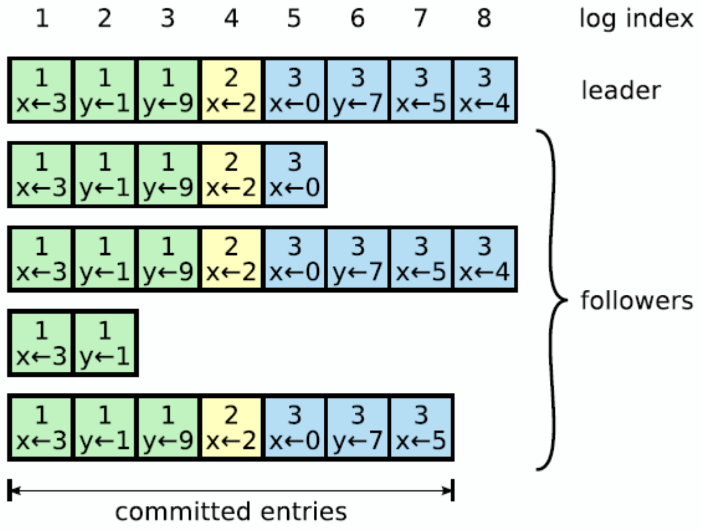
C is the new leader! and continue to send Heartbeats to the other nodes

Log Replication

Since we talk about consensus, what happened with the replicated logs?

When a leader get a command from a client then **appends it to its log as a new entry**, then issues AppendEntries RPCs **in parallel to each of the other servers** to replicate the entry. When the entry has been safely replicated, the leader applies the entry to its state machine and returns the result of that execution to the client.

In case of a follower failure, then the leader **retries** until **all followers** eventually store **all log entries.** So the leader decides when it is safe to apply a log entry (committed) to the state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers.



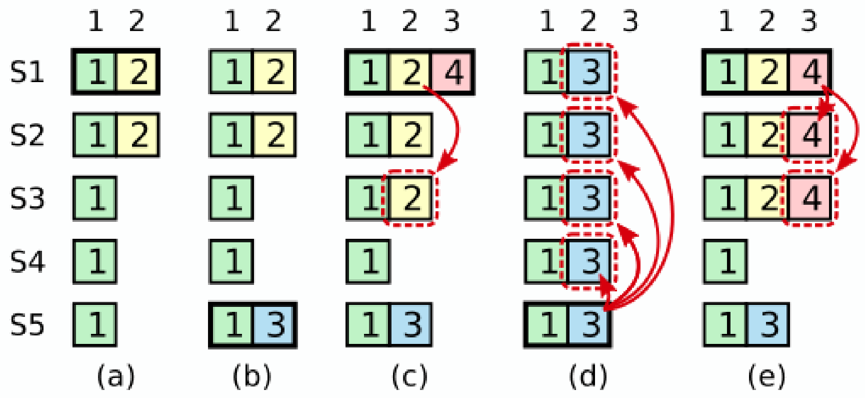
Log Replication (Cont.)

What happen when leader crushes?

When leader crashes can leave the logs **inconsistent!**. In our current followers(possible Candidates/Leaders) may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both.
Raft can handle these inconsistencies by forcing the followers' logs to duplicate leader's. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log.

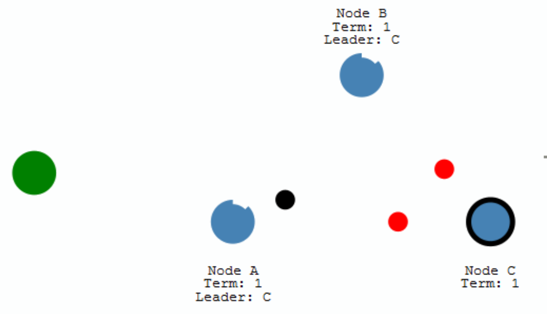
To bring a follower's log into consistency with its own is described to the next steps:

1. The leader find the latest log that they agree
2. Delete any entries in the follower's log after that point
3. Send the follower all of the leader's entries after that point.

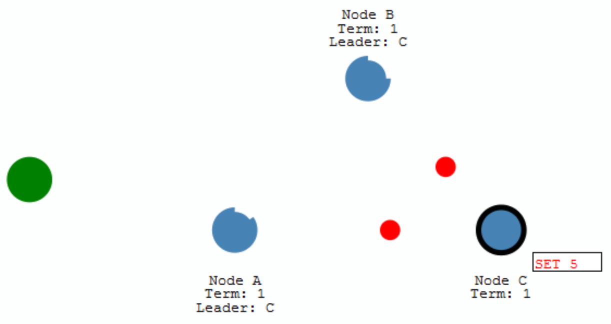


Log Replication Example

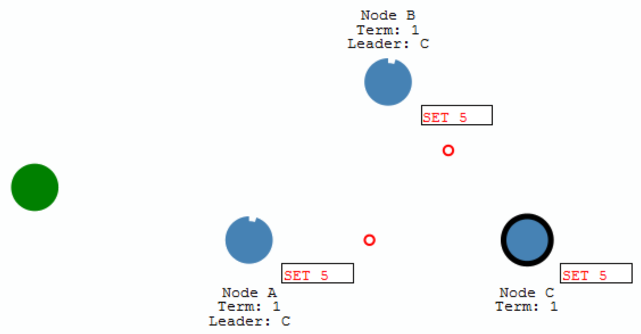
Let’s take a quick example, with normal replication during execution



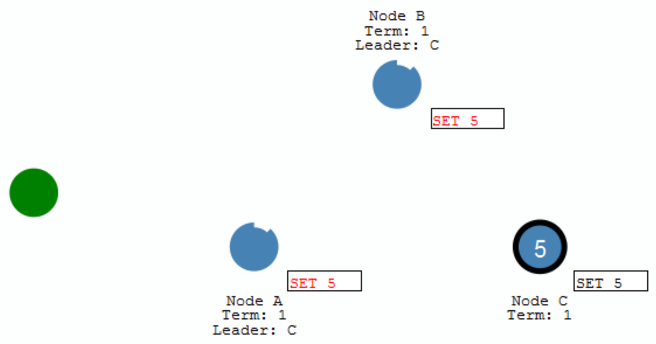
A client send a request to leader while he send Heartbeats to other nodes.



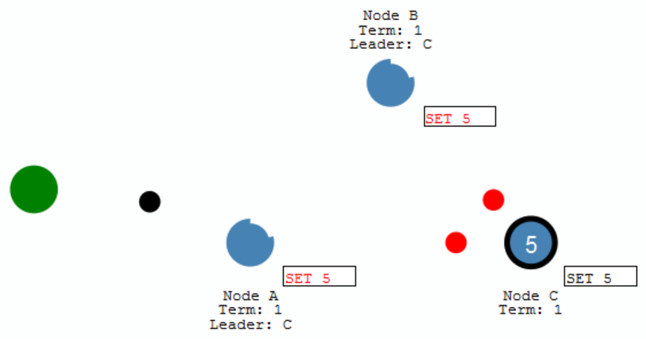
Leader get the request (command) and replicate it to other nodes



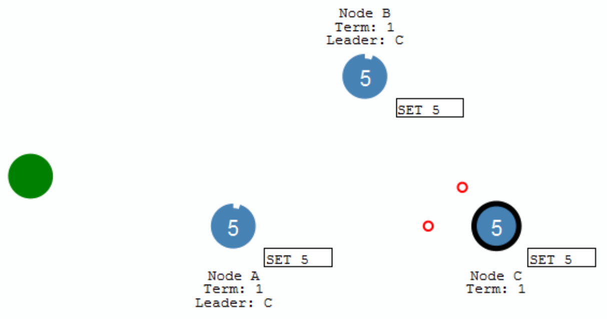
Nodes store the command in their logs



They response to the leader and he commit the result to it’s log



Leader response to client. Also sends the final commit to other nodes



They finally commit the command on their log

Safety

The mechanisms so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. We can make it by adding the following properties:

1. **Election restriction**
   The new leader need contain all the committed entries from previous terms on the moment of its election, without the need to transfer those entries to the leader.
2. **Committing entries from previous terms**
   Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's current term are committed by counting replicas. Once an entry from the current term has been committed in this way, then all prior entries are committed.
3. **Safety argument**
   If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. At the time a server applies a log entry to its state machine, its log must be identical to the leader's log up through that entry and the entry must be committed. Also Raft requires servers to apply entries in log index order. Combined with the State Machine Safety Property, this means that all servers will apply exactly the same set of log entries to their state machines, in the same order.

Safety (Cont.)

4. **Follower and candidate crashes**

   If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. If the crashed server restarts, then the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts

5. **Timing and availability**

   Raft will be able to elect and maintain a steady leader as long as the system satisfies the following *timing requirement*:

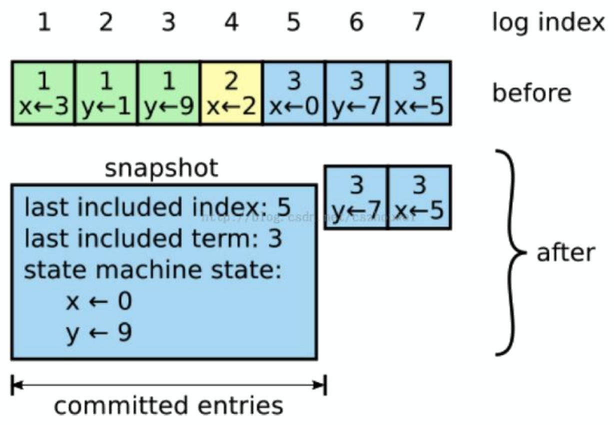
   $broadcastTime \ll electionTimeout \ll MTBF$

   In this inequality *broadcastTime* is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses, *electionTime-out* is the election timeout and *MTBF* is the average time between failures for a single server.

Snapshots

Raft's log grows during normal operation to incorporate more client requests, but in a practical system, it cannot grow without bound. As the log grows longer, it occupies more space and takes more time to replay.

To avoid that behavior Raft use **Snapshots**. In snapshotting, the entire current system state is written to a *snapshot* on stable storage, then the entire log up to that point is discarded.



The base idea behind Raft's snapshotting is that each server takes snapshots independently, covering just the committed entries in its log. Most of the work consists of the state machine writing its current state to the snapshot (and some metadata like *lastIndex*).

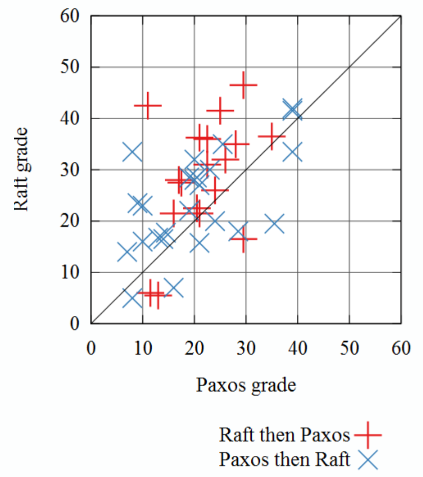
Although servers normally take snapshots independently, the leader must occasionally send snapshots to followers that lag behind. This happens when the leader has already discarded the next log entry that it needs to send to a follower.

Evaluation

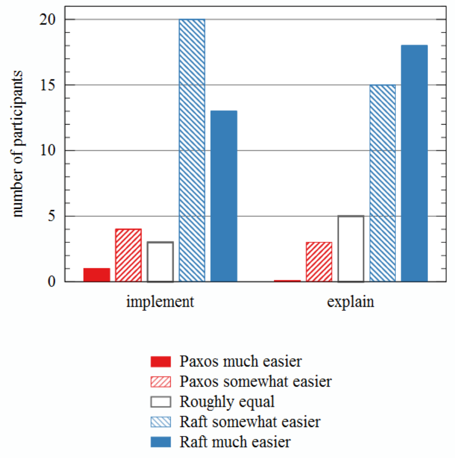
1. Understandability

To measure Raft's understandability relative to Paxos, conducted an experimental study using upper-level undergraduate and graduate students in an Advanced Operating Systems course at Stanford University and a Distributed Computing course at U.C. Berkeley.

The following are the results:



A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.



Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

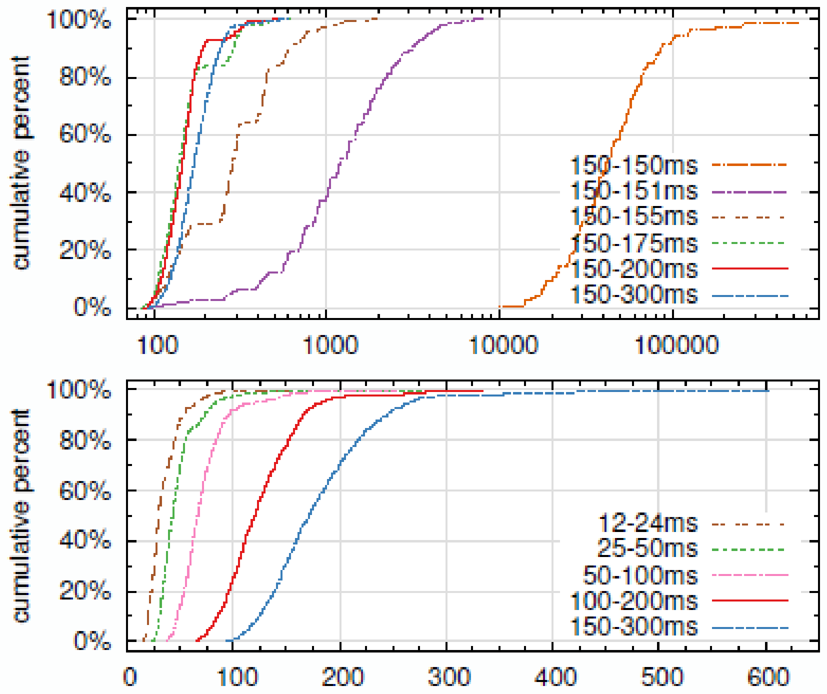
Evaluation (Cont.)

2 Correctness

Log Completeness Property mechanically proven using the TLA proof system. This proof relies on invariants that have not been mechanically checked.
An informal proof written of the State Machine Safety property which is complete and relatively precise

3 Performance

Raft's performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages.
To measure the performance of Raft's leader election algorithm and answer two questions. If the election process converge quickly (top graph) and what is the minimum downtime that can be achieved after leader crashes (bottom graph).



Why not Paxos

Paxos is a family of protocols for solving consensus in a network of unreliable processors. The basic problem with Paxos is that there is a big gap between description of Paxos and the real needs of a system. The final system will be based on an not proved system!

Below are the reasons that led to build a new algorithm:

1. **Understandability**
   Paxos is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this is difficult to develop intuitions about the single-decree protocols works. In multi-Paxos the complexity is increased.
2. **Bad foundation**
   It contains a not good foundation for building practical implementations. One reason is that there is no widely agreed upon algorithm for multi-Paxos. Lamport's descriptions are mostly about single-decree Paxos, he sketched possible approaches to multi-Paxos, but many details are missing.

Conclusion

Raft is one of the simpler to implement, partially asynchronous, consensus schemes. It's passively replicated and with small modifications can be used to create a fault tolerant log or state machine.. Even if there are similarities between Raft and other algorithms of that category there are many differences in many levels of their functionality.

Although is a new algorithm and need improvement to issues like performance (e.g. snapshot transferring) it become known very quickly and acquired great community (github projects). Databases like Kudu(implemented based on Raft)

Thank you