Raft FAQ

Q: Does Raft sacrifice anything for simplicity?

A: Raft gives up some performance in return for clarity; for example:

* Every operation must be written to disk for persistence; performance probably requires batching many operations into each disk write.

* There can only usefully be a single AppendEntries in flight from the leader to each follower: followers reject out-of-order AppendEntries, and the sender's nextIndex[] mechanism requires one-at-a-time. A provision for pipelining many AppendEntries would be better.

* The snapshotting design is only practical for relatively small states, since it writes the entire state to disk. If the state is big (e.g. if it's a big database), you'd want a way to write just parts of the state that have changed recently.

* Similarly, bringing recovering replicas up to date by sending them a complete snapshot will be slow, needlessly so if the replica already has an old snapshot.

* Servers may not be able to take much advantage of multi-core because operations must be executed one at a time (in log order).

These could be fixed by modifying Raft, but the result might have less value as a tutorial.

Q: Is Raft used in real-world software, or do companies generally roll their own flavor of Paxos (or use a different consensus protocol)?

A: There are several real-world users of Raft: Docker (https://docs.docker.com/engine/swarm/raft/), etcd (https://etcd.io), and MongoDB. Other systems said to be using Raft include CockroachDB, RethinkDB, and TiKV. Maybe you can find more starting at http://raft.github.io/

On the other hand, my impression is that most state-machine replication systems are based on the Multi-Paxos and Viewstamped Replication protocols.

Q: What is Paxos? In what sense is Raft simpler?

A: There is a protocol called Paxos that allows a set of servers to agree on a single value. While Paxos requires some thought to understand, it is far simpler than Raft. Here's an easy-to-read paper about Paxos:

  http://css.csail.mit.edu/6.824/2014/papers/paxos-simple.pdf

However, Paxos solves a much smaller problem than Raft. To build a real-world replicated service, the replicas need to agree on an indefinite sequence of values (the client commands), and they need ways to efficiently recover when servers crash and restart or miss messages. People have built such systems with Paxos as the starting point; look up Google's Chubby and Paxos Made Live papers, and ZooKeeper/ZAB. There is also a protocol called Viewstamped Replication; it's a good design, and similar to Raft, but the paper about it is hard to understand.

These real-world protocols are complex, and (before Raft) there was not a good introductory paper describing how they work. The Raft paper, in contrast, is relatively easy to read and fairly detailed.

That's a big contribution.

Whether the Raft protocol is inherently easier to understand than something else is not clear. The issue is clouded by a lack of good descriptions of other real-world protocols. In addition, Raft sacrifices performance for clarity in a number of ways; that's fine for a tutorial but not always desirable in a real-world protocol.

Q: How long had Paxos existed before the authors created Raft?

A: Paxos was invented in the late 1980s. Raft was developed around 2012.

Raft closely resembles a protocol called Viewstamped Replication, originally published in 1988. There were replicated fault-tolerant file servers built on top of Viewstamped Replication in the early 1990s, though not in production use.

A bunch of real-world systems are derived from Paxos: Chubby, Spanner, Megastore, and Zookeeper/ZAB. Starting in the early 2000s big web sites and cloud providers needed fault-tolerant services, and Paxos was more or less re-discovered at that time and put into production use.

Q: How does Raft's performance compare to Paxos in real-world applications?

A: The fastest Paxos-derived protocols are probably faster than Raft as described in the paper; have a look at ZAB/ZooKeeper and Paxos Made Live. On the other hand, etcd3 (using Raft) claims to have achieved better performance than zookeeper and many Paxos-based implementations (https://www.youtube.com/watch?v=hQigKX0MxPw).

There are situations where Raft's leader is not so great. If the datacenters containing replicas and clients are distant from each other, people sometimes use agreement protocols derived from original Paxos. The reason is that Paxos has no leader; any replica can start an agreement; so clients can talk to the replica in their local datacenter rather than having to talk to a leader in a distant datacenter. ePaxos is an example.

Q: Why are we learning/implementing Raft instead of Paxos?

A: We're using Raft in 6.824 because there is a paper that clearly describes how to build a complete replicated service using Raft. I know of no satisfactory paper that describes how to build a complete replicated server system based on Paxos.

Q: Are there systems like Raft that can survive and continue to operate when only a minority of the cluster is active?

A: Not with Raft's properties. But you can do it with different assumptions, or different client-visible semantics. The basic problem is split-brain -- the possibility of multiple diverging copies of the state, caused by multiple subsets of the replicas mutating the state without being aware of each other. There are two solution approaches that I know of.

If somehow clients and servers can learn exactly which servers are live and which are dead (as opposed to live but unreachable due to network failure), then one can build a system that can function as long as one is alive, picking (say) the lowest-numbered server known to be alive. However, it's hard for one computer to decide if another computer is dead, as opposed to the network losing the messages between them. One way to do it is to have a human decide -- the human can inspect each server and decide which are alive and dead.

The other approach is to allow split-brain operation, and to have a way for servers to reconcile the resulting diverging state after partitions are healed. This can be made to work for some kinds of services, but has complex client-visible semantics (usually called "eventual consistency"). Have a look at the COPS, FuzzyLog, and Bitcoin papers which are assigned later in the course.

Q: In Raft, the service which is being replicated is not available to the clients during an election process. In practice how much of a problem does this cause?

A: The client-visible pause seems likely to be on the order of a tenth of a second. The authors expect failures (and thus elections) to be rare, since they only happen if machines or the network fails. Many servers and networks stay up continuously for months or even years at a time, so this doesn't seem like a huge problem for many applications.

Q: Are there other consensus systems that don't have leader-election pauses?

A: There are versions of Paxos-based replication that do not have a leader or elections, and thus don't suffer from pauses during elections. Instead, any server can effectively act as leader at any time. The cost of not having a leader is that more messages are required for each agreement.

Q: How are Raft and VMware FT related?

A: Raft has no single point of failure, while VMware FT does have a single point of failure in the form of the test-and-set server. In that sense Raft is fundamentally more fault-tolerant than VMware FT. One could fix this by implementing FT's test-and-set server as a replicated service using Raft or Paxos.

VMware-FT can replicate any virtual machine guest, and thus any server-style software. Raft can only replicate software designed specifically for replication for Raft. For such software, Raft would likely be more efficient than VMware-FT.

Q: Why can't a malicious person take over a Raft server, or forge incorrect Raft messages?

A: Raft doesn't include defenses against attacks like this. It assumes that all participants are following the protocol, and that only the correct set of servers is participating.

A real deployment would have to keep out malicious attackers. The most straightforward option is to place the servers behind a firewall to filter out packets from random people on the Internet, and to ensure that all computers and people inside the firewall are trustworthy.

There may be situations where Raft has to operate on the same network as potential attackers. In that case a good plan would be to authenticate the Raft packets with some cryptographic scheme. For example, give each legitimate Raft server a public/private key pair, have it sign all the packets it sends, give each server a list of the public keys of legitimate Raft servers, and have the servers ignore packets that aren't signed by a key on that list.

Q: The paper mentions that Raft works under all non-Byzantine conditions. What are Byzantine conditions and why could they make Raft fail?

A: "Non-Byzantine conditions" means that the servers are fail-stop: they either follow the Raft protocol correctly, or they halt. For example, most power failures are non-Byzantine because they cause computers to simply stop executing instructions; if a power failure occurs, Raft may stop operating, but it won't send incorrect results to clients.

Byzantine failure refers to situations in which some computers execute incorrectly, because of bugs or because someone malicious is controlling the computers. If a failure like this occurs, Raft may send incorrect results to clients.

Most of 6.824 is about tolerating non-Byzantine faults. Correct operation despite Byzantine faults is more difficult; we'll touch on this topic at the end of the term.

Q: In Figure 1, what does the interface between client and server look like?

A: Typically an RPC interface to the server. For a key/value storage server such as you'll build in Lab 3, it's Put(key,value) and Get(value) RPCs. The RPCs are handled by a key/value module in the server, which calls Raft.Start() to ask Raft to put a client RPC in the log, and reads the applyCh to learn of newly committed log entries.

Q: What if a client sends a request to a leader, but the leader crashes before sending the client request to all followers, and the new leader doesn't have the request in its log? Won't that cause the client request to be lost?

A: Yes, the request may be lost. If a log entry isn't committed, Raft may not preserve it across a leader change.

That's OK because the client could not have received a reply to its request if Raft didn't commit the request. The client will know (by seeing a timeout or leader change) that its request may have been lost, and will re-send it.

The fact that clients can re-send requests means that the system has to be on its guard against duplicate requests; you'll deal with this in Lab 3.

Q: If there's a network partition, can Raft end up with two leaders and split brain?

A: No. There can be at most one active leader.

A new leader can only be elected if it can contact a majority of servers (including itself) with RequestVote RPCs. So if there's a partition, and one of the partitions contains a majority of the servers, that one partition can elect a new leader. Other partitions must have only a minority, so they cannot elect a leader. If there is no majority partition, there will be no leader (until someone repairs the network partition).

Q: Suppose a new leader is elected while the network is partitioned, but the old leader is in a different partition. How will the old leader know to stop committing new entries?

A: The old leader will either not be able to get a majority of successful responses to its AppendEntries RPCs (if it's in a minority partition), or if it can talk to a majority, that majority must overlap with the new leader's majority, and the servers in the overlap will tell the old leader that there's a higher term. That will cause the old leader to switch to follower.

Q: When some servers have failed, does "majority" refer to a majority of the live servers, or a majority of all servers (even the dead ones)?

A: Always a majority of all servers. So if there are 5 Raft peers in total, but two have failed, a candidate must still get 3 votes (including itself) in order to elected leader.

There are many reasons for this. It could be that the two "failed" servers are actually up and running in a different partition. From their point of view, there are three failed servers. If they were allowed to elect a leader using just two votes (from just the two live-looking servers), we would get split brain. Another reason is that we need the majorities of any two leader to overlap at at least one server, to guarantee that a new leader sees the previous term number and any log entries committed in previous terms; this requires a majority out of all servers, dead and alive.

Q: What if the election timeout is too short? Will that cause Raft to malfunction?

A: A bad choice of election timeout does not affect safety, it only affects liveness.

If the election timeout is too small, then followers may repeatedly time out before the leader has a chance to send out any AppendEntries. In that case Raft may spend all its time electing new leaders, and no time processing client requests. If the election timeout is too large, then there will be a needlessly large pause after a leader failure before a new leader is elected.

Q: Why randomize election timeouts?

A: To reduce the chance that two peers simultaneously become candidates and split the votes evenly between them, preventing anyone from being elected.

Q: Can a candidate declare itself the leader as soon as it receives votes from a majority, and not bother waiting for further RequestVote replies?

A: Yes -- a majority is sufficient. It would be a mistake to wait longer, because some peers might have failed and thus not ever reply.

Q: Can a leader in Raft ever stop being a leader except by crashing?

A: Yes. If a leader's CPU is slow, or its network connection breaks, or loses too many packets, or delivers packets too slowly, the other servers won't see its AppendEntries RPCs, and will start an election.

Q: When are followers' log entries sent to their state machines?

A: Only after the leader says that an entry is committed, using the leaderCommit field of the AppendEntries RPC. At that point the follower can execute (or apply) the log entry, which for us means send it on the applyCh.

Q: Should the leader wait for replies to AppendEntries RPCs?

A: The leader should send the AppendEntries RPCs concurrently, without waiting. As replies come back, the leader should count them, and mark the log entry as committed only when it has replies from a majority of servers (including itself).

One way to do this in Go is for the leader to send each AppendEntries RPC in a separate goroutine, so that the leader sends the RPCs concurrently. Something like this:

```
  for each server {
    go func() {
      send the AppendEntries RPC and wait for the reply
      if reply.success == true {
        increment count
        if count == nservers/2 + 1 {
          this entry is committed
        }
      }
    } ()
  }
```

Q: What happens if a half (or more) of the servers die?

A: The service can't make any progress; it will keep trying to elect a leader over and over. If/when enough servers come back to life with persistent Raft state intact, they will be able to elect a leader and continue.

Q: Why is the Raft log 1-indexed?

A: You should view it as zero-indexed, but starting out with one entry (at index=0) that has term 0. That allows the very first AppendEntries RPC to contain 0 as PrevLogIndex, and be a valid index into the log.

Q: When network partition happens, wouldn't client requests in minority partitions be lost?

A: Yes, only the partition with a majority of servers can commit and execute client operations. The servers in the minority partition(s) won't be able to commit client operations, so they won't reply to client requests. Clients will keep re-sending the requests until they can contact a majority Raft partition.

Q: Is the argument in 5.4.3 a complete proof?

A: 5.4.3 is not a complete proof. Here are some places to look:

http://ramcloud.stanford.edu/~ongaro/thesis.pdf
http://verdi.uwplse.org/raft-proof.pdf

Q: Are there any limitations to what applications can be built on top of Raft?

A: I think that in order to fit cleanly into a replicated state machine framework like Raft, the replicated service has to be self-contained -- it can have private state, and accept commands from clients that update the state, but it can't contact outside entities without special precautions. If the replicated application interacts with the outside world, the outside world has to be able to deal correctly with repeated requests (due to replication and replay of logs after reboot), and it has to never contradict itself (i.e. it has to be careful to send exactly the same answer to all replicas, and to all re-executions of log entries). That in turn seems to require that any outside entity that a Raft-based application contacts must itself be fault-tolerant, i.e. probably has to use Raft or something like it. That's fairly limiting.

As an example, imagine a replicated online ordering system sending credit card charging requests to some external credit card processor service. That external processor will see repeated requests (one or more from each replica). Will it respond exactly the same way to each request? Will it charge the credit card more than once? If it does do the right thing, will it still do the right thing if it crashes and reboots at an awkward time?