**Adding Scalability to Understandable Consensus**

Isaac Smead

*Computer Science Department*

*San Diego State University*

[*ismead@sdsu.edu*](mailto:ismead@sdsu.edu)

Abstract

Raft is a consensus algorithm that promotes understandability as its primary distinguishing feature. Most commonly used frameworks in cluster computing use a Praxos based approach to consensus. Raft is typically presented as an easy to understand alternative to the Praxos algorithm. All literature to date on Raft discusses its use in a small cluster of servers to maintain a shared state machine. This paper explores the application of Raft in a broader context of distributed peer to peer systems. We demonstrate the feasibility of using Raft to provide consensus over a larger scale environment and present a proposal of scaling Raft for systems with a large number of nodes.

# 1. Introduction

Ongaro and Ousterhout [1] developed Raft on the premise that the Praxos consensus algorithm is notoriously difficult to understand. Their criticism of Praxos extends to demonstrate a large rift between the core algorithm and its practical implementation. Praxos based systems are widely used by industry, including giants such as Google and Microsoft. These implementations, however are only loosely based on Praxos and contain a great deal of deviation from the core algorithm in order to achieve required functionality [1], [2]. Under this tenant, Ongaro and Ousterhout [1] make the case that Raft is a superior alternative to Praxos for building real world systems.

Since its introduction in 2014 Raft has gained popularity in both the academic community and with industry. It is covered in distributed systems courses at several top universities including MIT, Princeton and Stanford. There are at least 85 implementations of Raft on GitHub [3]. Apache Hydrabase and Docker are two of many industry-level programs utilizing the Raft protocol [4], [5].

Despite the growing popularity if Raft, we could find little discussion of its application outside of cluster computing. Howard [6] suggests future research could include diverse network topologies and byzantine fault tolerance. We agree, but also note that little to no attention has been given to studying the feasibility of using Raft in a large-scale system. Ongaro [1] briefly discusses limitations to Raft due to bottleneck, gives no attention to a possible solution. Conversely, he assumes that any modification to reduce bottleneck would adversely Raft’s primary objective of understandability and add a large degree of complexity. We respectfully disagree with this assumption and propose a solution to reduce bottleneck while maintaining Raft’s core tenants of understandability and simplicity.

The remainder of the paper is organized as follows: First we will discuss the Raft algorithm in greater detail, focusing on design attributes which incur bottleneck. Next, we will present our solution to reduce bottleneck. We then will detail our raft implementation and test environment. Finally we will present the result of our testing and discuss the implications.

# 2. Raft architecture and bottleneck

As with many consensus algorithms, Raft provides a set of rules and messages which allow a group of nodes to maintain a shared state machine. In Raft the state machine is kept up to date using a replicated log of actions. Once a log entry has been agreed upon by a majority of nodes the action can be applied to the state machine on each node. The Raft algorithm by design is built around the concept of a strong leader. Ongaro and Ousterhout [1] contend that having a strong leader facilitates their primary goal of understandability. This leader is the single point of entry for all actions and communicates directly with each node in order to maintain log replication. For simplicity a single RPC, *AppendEntries****,*** is used by the leader to maintain the replicated state. This RPC is continually broadcast at a frequency tailored to the network environment (typically 10 – 100 ms) [1], [2]. In the absence of new commands, an empty ***appendEntries*** is sent in order to maintain heartbeat.

Since all actions and associated coordination between nodes flow through a single leader it is easy to see how this design can be prone to bottleneck. Indeed, there are various ways in which a leader-based design can constrain throughput. One such limitation could be due to a higher volume of actions, or more complex actions than the leader can handle. Techniques such as partitioning the state into shards exist to mitigate problems of this sort. However, these techniques are highly complicated and beyond the scope of this research. Also, such approaches would almost certainly reduce understandability, as Ongaro feared.

A second, more approachable, form of bottleneck occurs because the leader is alone responsible for communicating actions in the replicated log among all nodes. In a Raft cluster of N nodes, the minimum number of messages the leader must send or receive in order to commit an action to the replicated log is . This may not be a problem in a system of only a few nodes, but it’s clear that Raft’s strong leader design would be a limiting factor in a system of hundreds or thousands of nodes.

Our challenge then is to reduce the number of messages sent and received by the leader without major modification to the Raft algorithm and without significant impact on understandability. Protocols such as gossip could reduce the workload required by the leader to notify all nodes of an action. However this would only partly solve the problem, because the leader would still need to receive confirmation separately from all nodes. One might suggest that nodes might also be able to communicate their responses amongst each other before replying to the leader. Doing this they would be able to send a small number of reply messages to the leader that contain an aggregate of the responses from all nodes.

It seems clear that any approach that involves random inter-node communication will likely introduce an un-acceptable level of complexity. Additionally, such methods have the potential to adversely impact Raft’s fault tolerance model. Raft does not require that message delivery is guaranteed, or that messages are time-bound. It does however use a timeout mechanism to detect a failed leader. In practice the addition of intra-node communication would likely require the adjustment of Raft’s timing parameters, and perhaps lead to more un-necessary leader changes. With these complications in mind it is understandable that Ongaro was skeptical of any measures to reduce bottleneck.

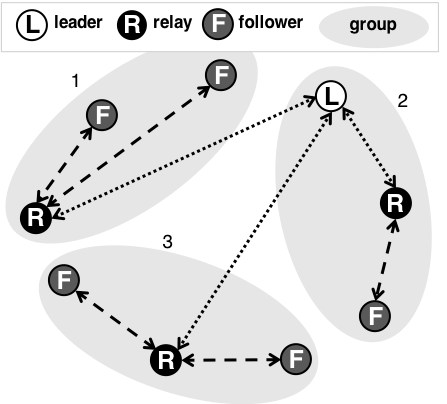
# 3. Reducing bottleneck in Raft through layering.

In this section we present our novel approach to the bottleneck problem through the introduction of a system hierarchy. Instead of viewing Raft’s strong leader concept as a limiting factor to throughput, we look at the leader as a key component in the approach to scalability. Our proposed system maintains a single strong leader, but we separate participating nodes into groups. Each group of nodes will have its own group-leader which we will call the “relay” node. We can assign each group a unique identifier in order to provide a mapping of nodes to groups and facilitate context changes discussed in later sections.

Each group will ideally contain a similar number nodes and function much like a Raft cluster. The distinction is that group’s relay serves as the conduit between members of the group and actual leader. Similarly, the leader and the several relays will constitute a core group and maintain replicated state as per the Raft protocol. We can think of the leader’s group as the first layer of the system. The aggregate of all other groups would make up the second layer. Conceptually we could add additional layers as necessary to achieve desired scale, but for simplicity sake we will limit our discussion to a two-layered system.

A simplified multi-layer Raft network with 3 groups and 9 total nodes is shown in Figure 1. The leader communicates bi-directionally with the relay node for each of the groups. The relay nodes in turn communicate bi-directionally with each of the follower nodes within their group. Leader-relay and relay-participant relationships both behave much like the leader-participant relationship in traditional Raft. Group membership is generally fixed and the mapping of nodes to groups is known by all nodes upon startup. This is necessary to support fault tolerance properties, as discussed in section 3.2. It is important to note also that the leader in Figure 1 is a member of group 2. In the event that of a leadership change, the node would continue participating in group 2 as a follower.

Although we add some complexity of depth with this layered approach the core Raft algorithm remains intact. The new role of relay can be thought of as a node that functions a leader in the context of its group and a follower in the context of the layer above. We assert that this layered approach maintains the understandability of Raft because groups at each layer can simply be viewed as an instance of the Raft protocol.



**Figure 1. multi-layer Raft organization**

## 3.1. Replicated state

Maintaining a replicated log in our multi-level approach is not much more complex than in the original algorithm. In order to ensure safety, the leader will not commit a log entry until it has been replicated in a majority of nodes. When the leader proposes a new entry to the log it will be propagated to all nodes via relay nodes. At this juncture relay nodes must wait some time in order to allow its followers to respond. In order not to slow progress of the entire system the relay should respond to the leader within a certain interval of time regardless of whether it has collected confirmations from all of its followers. Each relay will then reply to the leader with an aggregated confirmation for the followers which replied.

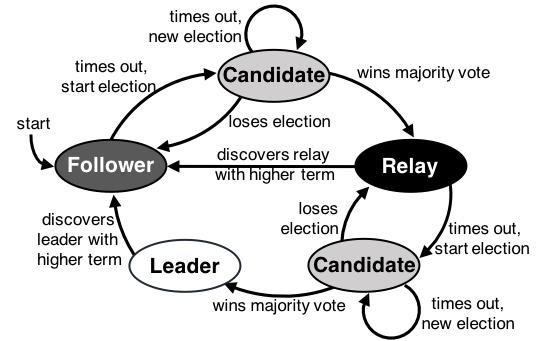
Relays will continue to accept confirmations from their followers after sending the aggregated confirmation message to the leader. Relays will then forward any late confirmations to the leader. In the event that a large number of followers are slow to respond the leader may not receive the majority of confirmations in order to commit nodes to commit the entry on the first round of responses from the relays. However, as late confirmations are forwarded on by relays the leader will eventually receive the quorum necessary to commit the entry.

## 3.2. Fault Tolerance

Arguably the most elegantly simple aspect of Raft is its approach to fault tolerance. As with most systems the failure of a participant doesn’t stop liveness until a majority of the participants have all failed. The most critical fault in a leader-based system such as Raft is the failure of the leader itself. In this case the system must be able to spontaneously recover by finding a new leader. Raft handles this through a random election process. All nodes are eligible candidates to take over leadership and each anticipate the failure of the leader by maintaining a timeout interval. Timeout intervals are randomly selected from a configurable range. This provides a high probability that a single node will notice a failure and become leader before its peers timeout and enter the competition to become leader. Ongaro [2] makes the case that this is more understandable and produces less edge cases than a hierarchal approach.

In our multilayered version we must consider the case of a failed relay and provide some slight modifications to the case of the failed leader. Figure 2 shows the sequence of roles a node can occupy throughout the election process for both relay and leader nodes. When a relay fails one or more of the follower nodes in the group will enter candidacy to become the new relay in accordance with Raft’s random election algorithm. Is the election is unsuccessful a new election round is repeatedly entered until a node succeeds in winning election. Upon successful election, the relay will assume responsibility of passing sending heartbeat messages and passing events from the leader to the group. Though there are various ways a newly elected relay could establish communication with the leader, a simple approach is to provide an abstraction layer between the leader and the relays. Instead of addressing messages to specific relays the leader would distribute a multicast to which the relay of each group would subscribe. When a new relay is elected it would simply replace its predecessor as the multicast subscriber.

Due to the fact that relays are the only nodes monitoring messages from the leader they will be the only nodes to notice a failed leader. Therefore, it must be a relay who assumes the role a failed leader. As shown if figure 2, election of a new leader proceeds in a similar manner as the election of relay. When a node is successfully elected and assumes the role of leader it will abandon its duties as relay. This will cause its group to be without a relay and force an election for a new relay. In the event that a failed leader was to recover it would notice that it is no longer leader and move to a follower position in its group. In order to reduce group down-time when a relay is elected as leader, the relay can designate a replacement relay from within the group. We explore this optimization further in section 4.



**Figure 2. State diagram of a multi-layer Raft node.**

## 3.2. Reduction of bottleneck

A homogeneous multi-layered Raft network of N nodes can be organized into groups, each with nodes. Under this architecture the leader would only need to send AppendEntries RPC to relay nodes in order propagate a log entry, and process responses from relay nodes in order to commit a transaction. Since each relay node must also communicate AppendEntries with its respective group, the total number of messages required to commit an entry is . As noted earlier, the traditional Raft algorithm requires messages in order to commit an entry. For the leader, complexity for committing log entries is reduced from to while maintaining the total number of messages in the system at . Considering that the AppendEntries RPC is continually sent on an interval to maintain heartbeat, our multi-tiered approach also serves to reduce baseline load on the host system since only heartbeat messages would need to be sent every interval.

# 4. Implementation and Testing

Though there exist many open source implementations of Raft, we chose to write our own for several reasons. First, it helped foster a more complete understanding of the algorithm. Second, we wanted to tailor our implementation to be easily modifiable as we work toward our goal of a multi-layer system. We chose to build our implementation in Node.js in order to avoid the complexity of a multi-threaded design. Node.js provides a completely event driven environment that allows for testing the behavior of multiple nodes within a single program. Nodes communicate via UDP datagram but this is abstracted through a generic connection interface which could easily be implemented in another protocol. All network communication occurs over the loopback adapter on a single network card. In order to more accurately model a distributed networking environment a network delay of with a mean of 20ms was applied. Actual delay values were randomly selected following from a Gaussian distribution using a 2ms standard deviation. Our Raft implementation and multi-layer modifications are available on github at <https://github.com/isaacsmead/raftjs>.

## 4.1. Leader and relay election

Given the limed time frame for this project we chose to focus our implementation and testing to the leader election aspect of the algorithm. As we mentioned in section 3.1, when the leader fails an election is held among the relay nodes to become the new leader. The victorious relay node’s group is now without a relay node due to the relay node’s promotion to leader. The participants of the group will notice they no longer have a relay and hold an election for a replacement. Though this solves the problem, the amount of time for the entire system to be restored is roughly double that of the traditional Raft algorithm because we must incur two election cycles before all nodes are once again participating in the network. Fortunately this additional delay can easily be mitigated.

Ongaro [1] devised an extension to Raft which allows for leadership transfer. This extension consists of a single RPC called *TimeoutNow.* If a leader wishes to relinquish its duties, it can send a TimeoutNow request to one of the nodes in the cluster before stepping down. Upon receipt of a TimeoutNow request a node will immediately hold and election to become leader. This enables the replacement of a leader with minimum down time while maintaining the safety of the system. In multi-layer Raft we can use the TimeoutNow command to reduce the delay incurred to elect a new relay after a relay is promoted to leader. When a relay is promoted to leader will send the TimeoutNow command to one of the follower nodes in its group. The recipient of the TimeoutNow command will immediately hold an election to become the new relay and likely assume the relay role in approximately 1.5 times the network delay.

## 4.1. Test results

In order to verify our downtime assumptions for multi-level raft we conducted a series of tests in which we would kill off the leader node and measure the time until a new leader was elected as well as the time until a new relay was elected in order to replace the relay that became leader. We ran these tests for both traditional Raft and our multi-layer algorithm. Ongaro [1] explored this measurement over a variety of timeout parameters and network latencies. He asserted that the ideal timeout parameters should be tailored to specific applications on the basis of network properties and application requirements. However, generally he suggests that a timeout window of 150-300 milliseconds with a heartbeat interval of 50-75 milliseconds should be ideal for most use cases. Since our primary goal of testing was to compare multi-level Raft with the performance of traditional Raft we chose to limit our testing to parameters the same or similar to those recommended by Ongaro.

Summarized results from our testing are shown in Figure 3. As a baseline we tested the standard Raft algorithm with clusters of 5, 9 and 15 nodes. We tested our multi-level algorithm with group sizes corresponding to the cluster sizes of the baseline tests. As such our multi-level tests consisted of 25, 81, and 225 nodes respectively. Testing traditional raft with larger clusters sizes would have been problematic because as the number of nodes increases the timeout window must be expanded in order to maintain liveness. Instead, by comparing a traditional Raft cluster of size N to a multi-layer network of size N2 we are able to use the same parameters for both. In our tests the timeout window for clusters of 5 and 9 was 150-300 milliseconds and 150-450 ms for the cluster of 15. All tests used a heartbeat interval of 75 ms and an average network delay of 20 ms.

# 5. Future research

Though there exist many open source implementations of Raft, we chose to write our own for several reasons. First, it helped foster a more complete understanding

# 6. References

[1] D. Ongaro, J. Ousterhout, “In Search of an Understandable Consensus Algorithm”, *2014 USENIX Annual Technical Conference*, USENIX, Philadelphia, PA, June 2014, pp. 305–320.

[2] D. Ongaro, “Consensus: Bridging Theory and Practice”, Stanford University, 2014.

[3] D. Ongaro, “The Raft Consensus Algorithm,” Raft Consensus Algorithm Website. [Online]. Available: https://raft.github.io/. [Accessed: 01-Apr-2018].

// not sure how to cite [4] and [5]. This is the only sort of documentation we could find of industry implementation of Raft

[4] https://issues.apache.org/jira/browse/HBASE-12476

[5] https://docs.docker.com/engine/swarm/raft/

[6] H. Howard, “ARC: Analysis of Raft Consensus”, Tech. Rep. UCAM-CL-TR-857, University of Cambridge, July 2014.

[7] H. Howard, M. Schwarzkopf, A. Madhavapeddy, J. Crowcroft, “Raft Refloated: Do We Have Consensus?”, *ACM SIGOPS Operating Systems Review*, Volume 49, no. 1, 2015, pp. 12-21.