Adding Scalability to Understandable Consensus

Isaac Smead
Computer Science Department
San Diego State University
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 any consensus algorithm, 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. This leader is the single point of entry for all actions and communicates directly with each node in order to maintain log replication. Ongaro and Ousterhout [1] contend that having a strong leader facilitates their primary goal of understandability.

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 2*N. 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 messages 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 sub-leader which we will call the "coordinator". 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 small number nodes and function much like a Raft cluster. The distinction is that group's coordinator serves as the conduit between members of the group and actual leader. Similarly, the leader and the several coordinators 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.

Although we add some complexity of depth with this layered approach the core Raft algorithm remains intact. The new role of coordinator 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.

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 coordinator nodes. At this juncture coordinator nodes must wait some time in order to allow its followers to respond. In order not to slow progress of the entire system the coordinator should respond to the leader within a certain interval of time regardless of whether it has collected confirmations from all of its followers. Each coordinator will then reply to the leader with an aggregated confirmation for the followers which replied.

Coordinators will continue to accept confirmations from their followers after sending the aggregated confirmation message to the leader. Coordinators 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 coordinators. However, as late confirmations are

forwarded on by coordinators 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.

With our multi layered version we must consider the case of a failed coordinator and provide some slight modifications to the case of the failed leader. When a coordinator fails it will be replaced by one of the nodes in its group using the Raft's random election algorithm. The newly elected leader now must assume responsibility of relaying events from the leader to its group. Though there are various ways this could be achieved in practice, a simple approach seems is to provide an abstraction layer between the leader and the coordinators. Instead of addressing messages to specific coordinators the leader would distribute a multicast to which the leader of each group would subscribe. When a new coordinator is elected it would simply replace its predecessor as the multicast subscriber.

Due to the fact that coordinators 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 coordinator who assumes the role a failed leader. When a node switches from a coordinator to a leader role it will cease its duties as coordinator. This will cause its group to be without a coordinate and force an election for a new coordinator. In the event that the original leader was to recover it would notice that it is no longer leader and move to a follower position in its group.

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, which results in non-realistic distributed networking environment. We plan to inject variable network latency and the occasional packet loss into our connection class in order to more accurately model the behavior of a real-world system.

4.1. Progress to date

Having been delayed by modifying our research topic to focus on Raft, we are not as far along in the testing and implementation phase as we would have hoped for this point. We have all the components in place for our implementation, but it is not yet complete. We have an easily configurable environment where we can instantiate a number of nodes, monitor their behavior and selectively kill them off as desired. Leader election functions properly as per the protocol. We have a persistent log built and unit tested, but have not yet achieved replication throughout the system.

Though there is much work left to do we are pleased that we have a partially working model and are not far from the point where we can begin testing our multi-layered version of the algorithm. Given the limited scope of this project and short time remaining we plan to focus our efforts testing and evaluating the leader / coordinator election aspect of our multi-layered version of Raft.

5. 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.