Scaling Raft For Varying Cluster Sizes

Drew Ripberger

November 13, 2018

Abstract

The increasing use of distributed systems in recent years brings about the need for more robust and scalable methods in which to reach consensus in a system. With the introduction of Raft, distributed consensus has become more widely available in used in the design of clusters, in many different scenarios. Though, as these clusters begin to increase in node size, they become increasingly less efficient at coming to consensus. We investigate ways to increase network throughput, nad propose methods to more effectively handle a growing number of nodes in a cluster, analyzing their effectiveness in a practical and realistic scenario.

1 Introduction

The Raft consensus algorithm originated to simplify the preexisting Paxos algorithm, while at the same time, solving the same core problem [6] with a similar efficiency. For years, Paxos had dominated distributed consensus. At its core it defined a way in which a system could come to agreement on a given state [2]. Though, Paxos can be incredibly hard to comprehend. Many papers have been published in an attempt to offer a clearer explanation as to how Paxos functions [3, 4], but it continues to be a difficult system to implement at a practical level.

Ultimately, these algorithms define a method for a system to agree on a state [1]. They work to build a fault tolerant approach to distributed systems, the *replicated state machine* [7]. In this context, a group of machines replicate a single state across themselves to create a fault tolerant system, that can handle the failure of a given node.

2 Scaling Distributed Systems

In practical applications, clusters of varying sizes are required. In some cases many nodes will be used, each replicating a small piece of data many times over. While in others, few nodes wil are used and larger chunks of data are replicated. Though in implementation, there are drawbacks to having a cluster with many nodes. As you continue to increase node number, various factors can lower a networks required time to reach consensus. Raft solves the consensus problem algorithmically, but let's, for example, take a look at a real world example where scaling comes into play.

We can imagine a large party of friends trying to decide where they want to go to eat for dinner. In this scenario in order to make an effective, and satisfying decision as to where the group should dine, each member of the group must be consulted. So the time in which it takes the entire group to come to an agreement increases as the size of the group increases.

In this large party, many more people will have to be consulted, and more options will have to be weighed before a choice is made. Though, compare this to just a few friends, who would be able to reach mutual agreement much faster, as they have less to consider, and fewer people that need to be taken into account before reaching a decision.

This principle is clearly demonstrated in distributed systems [5]. Adding more nodes to a cluster makes it more difficult for the network to handle faults and replicate its state. Demonstrated in Raft, the more *followers* in a system, the more heartbeats that have to be sent out by the *leader*, processed, responded to, and confirmed [6].

References

[1] Brand, D., and Zafiropulo, P. On communicating finite-state machines. *J. ACM 30*, 2 (Apr. 1983), 323–342.

[2] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[3] Lamport, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

[4] Mazieres, D. Paxos made practical. *Unpublished manuscript, Jan* (2007).

[5] Neuman, C. Scale in distributed systems. *Readings in Distributed Computing Systems* (1994).

[6] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[7] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv. 22*, 4 (Dec. 1990), 299–319.