Dinghy: Horizontally Scaling Raft Clusters

Drew Ripberger

November 15, 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 [14] 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 [10]. 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 [11, 12], 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 [8]. They work to build a fault tolerant approach to distributed systems, the *replicated state machine* [15]. 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 $n/2 - 1$ nodes. The essential goal of consensus in terms of the *replicated state machine* is to reach a *univalent* state, from any *multivalent* state. Such algorithms specifically order state changes, to ensure that when applied, that all result in the same state [9, 14]. Raft also works to correct, and right, any nodes in a cluster in contradicting states. It does this via counting `election terms`, demonstrated as such:

We define two nodes in a cluster, with two corresponding state machines, M and N. We also define a function, $T(S)$, of some arbitrary state machine S, that is its current `term`.

$$\text{Correct State} = \begin{cases} T(M) > T(N), & M \\ T(M) < T(N), & N \end{cases}$$

In order to keep some sense of order in the cluster, Raft keeps track of the number of leader elections that have occured with the `election term`. This is a system wide tally that is used to determine when a node may be behind or have conflicting information in its log. Many of these comparisons have to be made from node to node through heartbeat messages, that also act to check for leader liveness [14]. But, as one might imagine, as you try to include more nodes in a cluster, the number of checks that have to be propogated to ensure an effective system drastically increases.

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 [13]. 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 then confirmed [14].

2.1 Applications in Raft

Like all other algorithms of its class, Raft has similar problems scaling. Though it has a distrinctly unique set of problems. It should be noted that we can consider Raft generally *synchronous*. That is all nodes must reach a *univalent* state, before handling the replication of the next update to the log. Raft's use of a *leader*, requires all changes to the network state be processed through a single node, thereby essentially creating a synchronous cluster where in which *followers* have very little power to influence the cluster's state where there is a healthy *leader*. So when we discuss scaling in terms of a leader based methods we have to examine different properities of the algorithm.

3 Testing Consensus

The big draw of Raft is its practicallity. It has allowed many the opportunity to implement distributed consensus, as it was designed to be understandable. So with this is mind, we maintain a similar dogma when it comes to testing the algorithms scalability. We wanted to design a test that would accomplish two main goals: *(1)* stress test the network, its throughput, and its ability to maintain consensus; *(2)* realistically represent an application or use case that Raft might see in a real world system.

With these considerations, we first decided the most common use case for this method of distributed consensus. The official Raft website contains a useful list of implementations [7]. Given its capabilities, Raft is very commonly used for database replication [1–3, 6], so we decided to have our *replicated state machine* be small key-value store, essentially a replicated mapping.

We elected to use a popular open source library written in Go, `hashicorp/raft` [4]. Then we wrote a simple key value store *replicated state machine*, into a barebones example for greater efficiency [5]. We spun up a cluster of an increasing, odd, size, and run a stress test on the cluster. The key value store is initialized with 10,000 entries as shown:

```
key1:  0
key2:  0
...
key10000:  0
```

Then after initializing the state machine, we start issuing requests to the network for a change in state. We go one by one, requesting a change, and a subsequent read in the value, of one key-value pair. After every 100 requests made to the network, we attempt to simulate node failures. Our script evaluates each node in the cluster. The script randomly decides whether or not said node should *fail*. Independly, we give each node a 10% chance of failure. This process of node failure in our testing we call *Intentional Random Node Failure*. We then reconnect all failed nodes, after 1 request to the network has been made. By the end of one full routine, we have gone through a total of 100 *Intentional Random Node Failures*, and have updated 10,000 values in our replicated key-value store such that:

```
key1:  1
key2:  2
...
key10000:  10000
```

The metric we use to compare between tests is the time in which the this routine takes to complete. This should be fully evaluate Raft consensus via both throughput, and fault tolerance.

4 Base Performance

Of course before we look into some potentially beneficial alterations we could make to Raft, we must examine the base algorithms ability to scale.

5 Approaches To Scaling Raft

Now we begin to look at some of the possible methods to scale the Raft consensus algorithm, each with varying degree of complexity and success.

5.1 Non-Voting Servers

5.2 Dynamic Timeouts

5.3 Node Grouping

References

[1] Cockroachdb - the open source, cloud-native sql database. `https://github.com/cockroachdb/cockroach`. Accessed: 2018-11-15.

[2] Distributed reliable key-value store for the most critical data of a distributed system. `https://github.com/etcd-io/etcd`. Accessed: 2018-11-15.

[3] Distributed transactional key-value database, originally created to complement tidb. `https://github.com/tikv/tikv`. Accessed: 2018-11-15.

[4] Golang implementation of the raft consensus protocol. `https://github.com/hashicorp/raft`. Accessed: 2018-11-15.

[5] Lightweight and simplistic kv raft implementation using hashicorp-raft. `https://github.com/drewrip/dinghy`. Accessed: 2018-11-15.

[6] The open-source database for the realtime web. `https://github.com/rethinkdb/rethinkdb`. Accessed: 2018-11-15.

[7] Raft consensus algorithm. `https://raft.github.io`. Accessed: 2018-11-15.

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

[9] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.

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

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

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

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

[14] 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.

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