Dinghy: Horizontally Scaling Raft Clusters

Drew Ripberger

March 16, 2019

Abstract

With the introduction of Raft, distributed consensus has become more widely available in the use of cluster design. Different applications of distributed systems often require differing configurations, however, as these clusters begin to increase in the number of participating nodes, they become increasingly less efficient at coming to consensus. Current methods of scaling distributed systems generally implement some variation of data sharding, batch processing, or message coalescing. We investigate however, utilizing dynamically set timeouts and heartbeat intervals to increase network throughput, and propose methods to more effectively handle a growing number of nodes in a cluster, while analyzing their ability to increase the horizontal scalability of a Raft cluster.

1 Introduction

The Raft consensus algorithm originated to simplify the preexisting Paxos algorithm, while at the same time, solving the same core problem [19] 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 [14]. 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 [15, 16], 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 [10]. They work to build a fault tolerant approach to distributed systems, the *replicated state machine* [20]. 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 [13, 19]. 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`, where T_c is the correct `term` in the cluster.

$$T_c = \begin{cases} M, & T(M) > T(N) \\ N, & T(M) < T(N) \\ \varnothing, & T(M) = T(N) \end{cases}$$

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 [19]. 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 [17]. 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 to the *leader*, processed, responded to, and then confirmed [19].

Though we can also demonstrate the benefits of a larger cluster. A greater number of nodes in a cluster allows for greater fault tolerance. If we say there is a 25% chance that a node fails at some point, we can model the probability of a cluster reaching agreement as such:

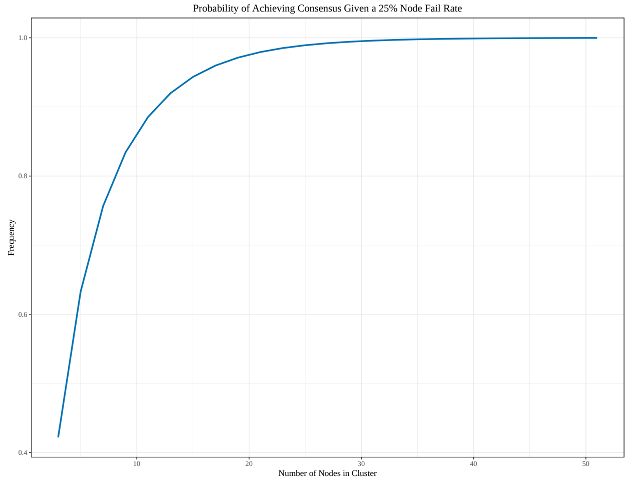
We define a function, $L(n)$, that is the probability a cluster of size n, can reach consensus given they have a 25% of failing. Essentially equivalent to the common *binomial cumulative density function.*

$$L(n) = 1 - \sum_{i=0}^{n/2-1} \binom{n}{i} (0.75)^i (0.25)^{n-i}$$

With this function, we model the probability given each cluster size:

$$\{(n, p) : n \in \{3, 5, 7, ..., 51\}, p = L(n)\}$$

Producing a distribution as such:



With this distribution we can see the theoretical benefits of a greater horizontal size. When increasing the number of servers, we initially see a great jump in the probability that the cluster reaches consensus. While this probability does increase, the servers quickly hit a point, in this instance at a size of about 30 servers, where the there are no observable benefits to fault tolerance.

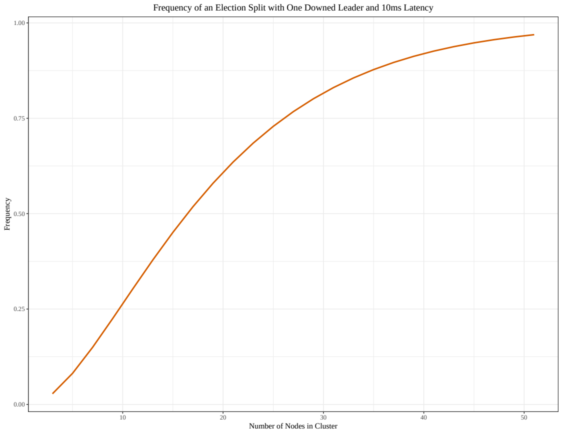
Also, as defined in David Ongaro's orginal thesis on Raft, we can theoretically calculate the probability of a cluster having a split vote [18]. These split votes contribute massively to the problems that go along with horizontal scaling.

We can simplify his orginal formula, and plug in some constants as such:

$$\sum_{k=c-1}^{s} \binom{s}{k} l^k (1-l)^{s-k}$$

$$\sum_{k=1}^{n} \binom{n}{k} .1^k (1-.1)^{n-k}$$

In this scenario the there is a latency of 10ms, only 2 nodes are splitting the vote, and no nodes are down. We can visualize this for a cluster size of n with the following:



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

2.1 Non-Voting Servers

One common way of increasing the capabilities of a large Raft cluster, is to change the voting status of additonal nodes. Usually when we discuss creating larger clusters, we mean creating a cluster of servers, each with full voting priveleges. But every one of these members all having full voting power, requires extra considerations in the cluster. When you are a *full* node, you must participate in log replication, as well as be consulted when changes need to be made to the replicated log. So a common solution to this requirement for being a *full* voting server, is making some of the members *non-voting* servers. This allows for the node to participate in the cluster, gaining the benefits of the log replication, but not bogging down the cluster by increasing the quorum size [2]. Though it should be noted that this solution does not in any way solve the presented problem. Demoting nodes to a non-voting configuration, does not assist with the fault tolerance of the cluster. Supose you spin up a cluster with 5 full Raft servers, and 100 non-voting servers. If even just 3 of the full nodes fail, the system can no longer make progress, even though we have allocated resources for running a total of 105 servers because, the majority of the voting nodes have failed.

2.2 Sharding

There is also a way of scaling writes and reads. Some databases divide up the data their are trying to replicate using a process called *sharding*. A great example of this process can be found in Google's Spanner database [11]. *Sharding* works as a type of distributed load balances to balance the store of data across many consensus groups.

2.3 Batching

Other databases work to alter the process of sending or receiving inter-node messages. The Calvin database works to group messages, and specially schedule them in order to maintain consistency, and scalability [21].

3 Dinghy Algorithm

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 when 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.

Dinghy works as an embedded algorithm within Raft. It runs side by side the core Raft *follower* routines, in order to actively maintain historical statistics on a node. In its most base form Dinghy attempts to get a sense of the environment that a given server is running in and adjust some of its previously set, static parameters, and adjust them to a more appropriate level for the situation. The Lifeguard system works in a similar fashion, but with differing goals, and approaches, using a step function to help stop false positives, whereas Dinghy uses specialized ping messages [12].

Originally, Dinghy was being designed to assess network latency on a node per node basis, each being allowed to alter their own timeout as they saw fit. This however challenges Rafts guarantee of progress, as it creates the potential for more frequent election splits, as well as promoting the asymmetric nature of the network. With some revisions, we were able to slightly adjust our original idea to ensure Rafts original guarantees. Dinghy functions as follows:

$$\delta : C \mapsto \mathbb{Z}$$

$$A = \{n \in C : n \text{ is returning heartbeat messages}\}$$

$$T_a = \frac{\sum x \in A}{|A|}$$

Now are new heartbeat timeout becomes the product of some *Dinghy Constant* D_k, and T_a. In our testing we used a constant of 20.

$$D_k T_a$$

This revised timeout is calculated every time a new leader node is elected, and piggybacks off of the AppendEntries RPC, adding a reviseTimeout property that tells all follower nodes to update their heartbeat timeout to the agreed upon duration.

4 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 stress test the networks throughput given a certain size, given that Raft is overwhelmingly used in creating fault tolerant databases [1, 4, 5, 8].

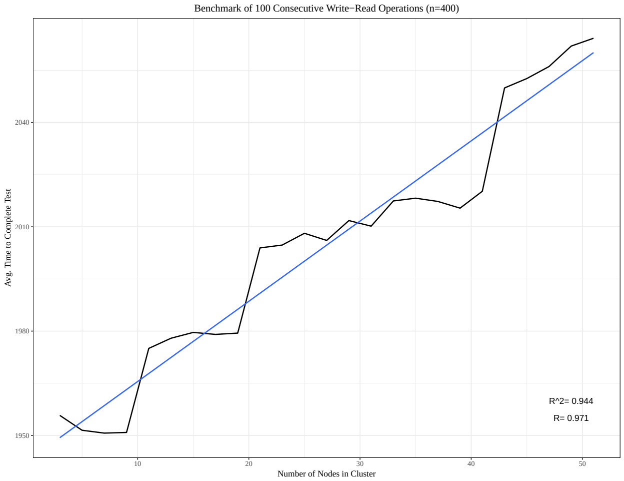
With these considerations, we first decided the most common use case for this method of distributed consensus. The official Raft website contains a useful list of implementations [9]. Though none of these would illustrate the problem that needs to be solved. One of the fundemental benefits of a consensus algorithm is their fault tolerance. So, we elected to simplify the problem down to its core component: how quickly could a cluster recover from a downed leader. We developed a simulation that would test how long a cluster of some size would recover. With this simulation written we then pitted our proposed algorithm Dinghy, against the vanilla version of Raft.

The code for this project is running, is called AvailSim. It is a variation on a test developed for the Dr. Ongaro's original Raft dissertation, written in Go [6]. We use this specific procedure in the testing of both our proposed algorithm, Dinghy, and the benchmarks of pure Raft. We also used a variation of testing and prototyping scripts in the process of development [3].

5 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. Using a simple implementation of Raft with the *hashicorp/raft* library, we started up clusters of varying sizes, and ran 100 successive write then read operations, and timed how long it took to complete [7].

As we see in this test, as we increase the number of nodes in a cluster, even without any extra difficulties like delays or node failures, consensus becomes harder to achieve. After this observation we can start to investigate improving



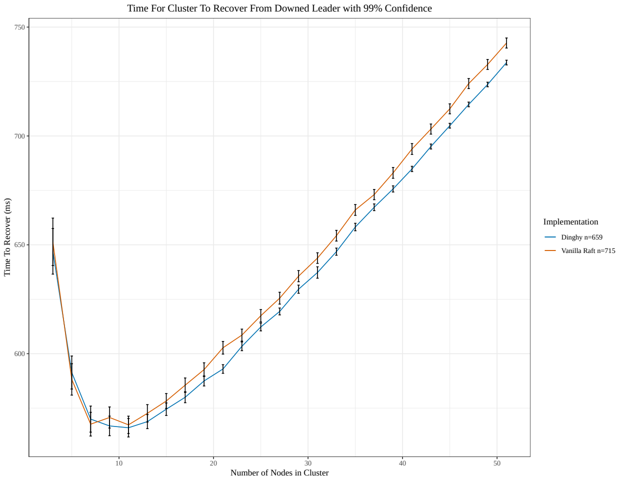
the rate at which the time it takes to reach distributed agreement.

6 Dinghy Performance

Testing the algorithm using the previously defined test.

As we see the small clusters in both algorithms have definitively higher recover times than those of the larger cluster. While this might seem counterintuitive at first, in a system this isolated and latency free, random timeouts can have larger effects on the quick recover of a cluster. Probabilistically this makes sense. A cluster will settle on a leader quite quickly after a node has recognized the leader to be down, and if this timeout is based on some random variable, the clusters with more nodes will have a greater probability of getting a smaller timeout that catches the downed leader quicker. Though with this consideration, we can quickly see the effects that larger cluster size have. The recover time of the cluster sky rocketed when more and more nodes were introduced. What we also see however is that the test with Dinghy taking consistently less time to recover than pure Raft.

From the generated 99% confidence intervals, we see that Dinghy consistently, and signficantly out performs pure Raft in terms of fault tolerance and its ability to scale. With this we can confidently say that Dinghy does in fact assist the Raft algorithm in scaling for larger cluster sizes, as the confidence interval for the difference betweent their recover times, does not include 0, which would indicate their equality in performance.



7 Conclusion

It becomes more and more difficult for pure Raft to achieve distributed agreement with more participants in the process. More messages have to be exchanged in order to perform the same operation, while a leader can only process communication from so many nodes at a time. We presented an algorithm that can be used in order to optimize the *heartbeat timeout* and *interval* of a node in order to allow for greater message throughput, and a quicker detection of faulty *leaders*.

The Dinghy algorithm uses the average of the elapsed message times, in order to gain a better idea of network latency. Using this measure, timeouts can be recalculated to allow for a greater efficiency in handling a growing number of messages. So in use with a larger cluster, Dinghy can be utilized to provide better horizontal scaling. In the future we hope to be able to combine this with other methods to achieve a well rounded approach to the scaling problem. Ideally in the near future we can make it more plausible to utilize an increasing number of nodes in the replication of data.

Time Taken To Recover After Downed Leader Node (ms) with 99% Confidence

Cluster Size	Vanilla Raft	Dinghy Raft	Improvement with Dinghy
3	651.37	647.03	4.34±15.10
5	588.19	591.35	-3.16±10.46
7	567.63	569.95	-2.32±8.08
9	570.70	566.82	3.88±6.60
11	567.32	565.99	1.34±5.81
13	572.61	568.80	3.81±5.13
15	578.28	574.51	3.77±4.45
17	585.55	579.99	5.55±4.15
19	592.69	587.49	5.21±3.88
21	602.72	592.97	9.75±3.52
23	608.53	603.38	5.15±3.40
25	617.42	612.26	5.16±3.36
27	625.49	619.40	6.09±3.15
29	635.61	629.62	5.99±3.17
31	643.91	637.29	6.62±3.58
33	654.19	646.85	7.33±2.96
35	666.01	658.18	7.84±3.01
37	673.07	667.24	5.83±2.81
39	683.04	675.68	7.36±2.86
41	694.01	684.85	9.16±2.77
43	703.16	695.15	8.01±2.59
45	712.46	704.71	7.76±2.54
47	724.06	714.48	9.58±2.54
49	732.84	723.64	9.20±2.51
51	742.69	733.71	8.98±2.55

References

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

[2] Consul enterprise enhanced read scalability. `https://www.consul.io/docs/enterprise/read-scale/index.html`. Accessed: 2019-1-24.

[3] Dinghy: Horizontally scaling raft clusters. `https://github.com/drewrip/dinghy`. Accessed: 2019-1-24.

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

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

[6] Go implementation of availsim raft test. `https://github.com/drewrip/availsim`. Accessed: 2019-3-15.

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

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

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

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

[11] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX Association, pp. 261–264.

[12] Dadgar, A., Phillips, J., and Currey, J. Lifeguard : Swim-ing with situational awareness. *CoRR abs/1707.00788* (2017).

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

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

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

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

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

[18] Ongaro, D. Consensus: Bridging theory and practice.

[19] 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.

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

[21] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 1–12.