Split-Brain Consensus

On A Raft Up Split Creek Without A Paddle

John Burke
jcburke@stanford.edu

Rasmus Rygaard
rygaard@stanford.edu

Suzanne Stathatos
sstat@stanford.edu

ABSTRACT

Consensus is critical for distributed systems. The consensus problem is simple with stable nodes and a reliable network, but in practice algorithms have to handle both failure modes. This paper introduces Timber, a system to explore how implementations of the Raft protocol [5] handle partitions and node failures. This application can be used with any implementation of the protocol and introduces network partitions and failures at user-prescribed intervals. We run Timber against two popular implementations of the protocol and discuss the results. We find that some implementations appear to tolerate network partitions, while others can drop more than 20% of writes under a slow, highly partitioned network.

1. INTRODUCTION

Modern software systems are often comprised of dozens of unreliable machines communicating over an asynchronous network. Messages in this system must necessarily have timeouts so that the system does not block indefinitely while waiting to hear from a failed machine. However, because a given message may be delayed an unbounded amount of time by the network, this scenario allows for network partitions: cases in which machines are believed to have failed due to message timeouts when in fact the messages have just been dropped or delayed by network problems. When designing and evaluating distributed systems protocols, it is important to consider whether the protocols′ guarantees of reliability, consistency, and fault tolerance can still be delivered in the face of such situations. Raft [5] is a consensus protocol by which distributed nodes can maintain one common replicated state machine. Raft claims to maintain consistency by using replicated logs and leader election. An illustration of Raft's replicated state machine architecture and consensus algorithm through leader election and log replication can be found below (Fig. 1).

It's one thing to describe how a consensus protocol works, and to describe all of the edge cases for handling faulty networks. It's another thing to see a protocol in action. This project builds Timber, a checking infrastructure that can plug into different Raft implementations (or, with a little work, arbitrary distributed storage systems) and find potential protocol errors. We implemented a distributed system across multiple EC2 instances, which collectively are called a cluster. We use techniques inspired by Jepsen [2] to tamper with network and node conditions. On each cluster of machines, we set up two Raft implementations: LogCabin (the C++ implementation written by the author of the RAFT paper) [4], and etcd/raft (used in CoreOS, written in Go) [6].

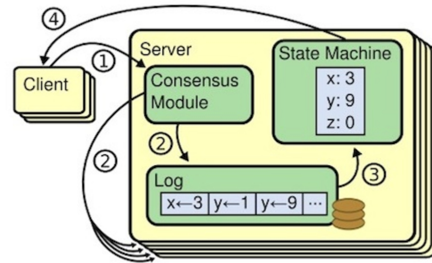

Figure 1: Raft uses replicated state machines to guarantee correctness. The state machines process identical sequences of commands from logs. Collectively, the machines elect a leader with the most up-to-date log to report back to the client issuing commands. In Timber's implementation, the replicated state machines run in an EC2 cluster. [5]

2. TIMBER IMPLEMENTATION

2.1 Platform Choice

In order to set up a distributed system, we needed a cluster of machines whose network we could control. To maintain control of the machines and their network conditions, we used EC2 instances. These instances were machines running Ubuntu 14.04 to take advantage of iptables. The machines are fairly low-powered 2.5 Ghz machines with 1 GB of RAM. We were mostly interested in the relative difference between the implementation and not absolute performance, so we opted for less powerful machines. Our setup is easily reproducible (steps to reproduce Timber for your own learning pleasure are in Timber's README) [1].

We created python scripts (launch_ec2.py) to launch N micro instances. These instances' public and private IP addresses

are saved locally. This master config file is later propogated to each of the nodes in the cluster to ensure that all nodes know each others' identities.

This cluster is set up as a variable number N nodes. For our implementation, we used $N = 5$ machines. According to the Raft protocol, then, $N = 2f + 1 = 5$ would allow for 2 node failures. These nodes can be partitioned (and therefore live, but become unable to talk to the rest of the cluster) or these nodes can be terminated.

After setting up our cluster, we install all of the dependencies needed for timber and for our remote procedure calls. We made sure that each machine could talk to every other machine in the cluster; finally, we made sure that each node could be identified and manipulated by its designated node name ("n1"..."n5"). We also keep a permanent backup copy of this information on each machine in /etc/hosts.

2.2 Controlling the Cluster

Timber provides users with a REPL. Users can execute commands to run against the entire cluster. They can use this to run individual commands to analyze how they affect Raft's implementation, or they can pass in files containing lists of commands to run. We also plan to implement a randomized torture test that will generate random partition and kill commands while sending a heavy write load to the cluster.

Timber uses Remote Procedure Calls (RPCs) to communicate with the cluster. These RPCs manipulate the iptables settings across machines to create and heal partitions. The source code is publicly available [1]. The network RPCs will be described in the following section.

2.2.1 Consistent Writes

We wanted to compare how our implementations of Raft operate with and without partitions. To observe this behavior, we run a simple set of transactions to write integers to a file. The steps follow.

1. A client calls *write_ints*
2. The Raft protocol is responsible for distributing the writes among nodes. When we write to one node, it is either a follower or a leader. If it is a leader, it will propogate the write to the other nodes in the cluster. If it is a follower, it will either forward the write to the leader or reject it and tell the client where the leader is (depending on the implementation). This is described in more detail in the original Raft paper [5].
3. The client keeps track of which nodes successfully wrote and which nodes failed. These measurements are listed when the process completes.

2.2.2 Partitions

We used UNIX's iptables command line tool to manipulate the network. This provided us a relatively painless way to prevent one machine from accepting traffic from or sending traffic to other machines, without requiring us to manipulate packets once they were received at the NIC.

So far, we have implemented four network action calls. Three of them pertain to partitioned machines. These are *partition*, *snub_nodes*, and *heal_cluster*. *Partition* splits the cluster in half. For example, in our cluster of 5 machines, calling partition creates two groups, *n1, n2* and *n3, n4, n5*. Clients identify which group they belong to and drop traffic from the opposite group. So, for example, nodes in group one run and attain the iptable configuration in Figure 2. *Snubnodes* splits the cluster into a group with the specified nodes and another group with the unspecified nodes. For *snubnodes 1 5* would make one partition with *n1, n5* and one partition with *n2, n3, n4*. *Heal_cluster* restores all of the nodes in the cluster (in both partitions) to their original network settings so that they accept traffic from all other nodes in the cluster. The fourth command, *slow*, is described below.

2.2.3 Delays

Distributed systems in the wild often face or have to handle slow nodes. In practice, TCP state machines let nodes "reliably" reconstruct packets received so that they are processed in order. TCP sockets guarantee that (without partitions or failed nodes) packets will not be dropped, duplicated or reordered. However, TCP does not provide a guarantee about timing. Therefore, a slow node and a faulty node have the potential to look the same to Raft. Furthermore, even if nodes are not slow enough to trigger heartbeat or election timeouts, slow nodes might expose windows of unsafe reads or writes that would be tough to expose otherwise. As an example, Kingsbury found in [3] that Postgres has a brief window where acknowledgements in two-phase commit are in flight and inconsistent results may be externalized. This window, however, is only feasible to expose if the network delays the acknowledgements.

Raft claims that slow servers do not affect the system, because as soon as a majority of nodes have responded to a round of remote procedure calls, the leader will apply its entry to a state machine, and the system can move to the next transaction. Followers lagging behind, then, will be able to update by having the leader retry Append-Entries RPC's indefinitely until all of the followers have up-to-date logs.

To test the functionality of this part of the Raft protocol, Timber supports a function call – *slow* – which adds delay to packets on each node. In the current implementation, the added delay follows a normal distribution with mean 50ms and variance 20ms. This has proved sufficient to expose network delay issues while avoiding too many leader elections because of election timeouts.

3. RESULTS AND DISCUSSIONS

3.1 Raft Implementations

We ran Timber against two popular Raft implementations. Although there are many Raft implementations available online, most of them prioritize the common case rather than fault tolerance. We therefore picked mature implementations that claimed to supported leader election and fault tolerance. Since the protocol is often closely tied to some outside system rather than a standalone consensus service, we decided to test the outside system that uses Raft internally instead of dissecting it to test the Raft implementation in isolation. This has the unfortunate side-effect that an in-

```
ubuntu@ip-172-31-36-51:~$ sudo iptables -A INPUT -s n3 -j DROP
ubuntu@ip-172-31-36-51:~$ sudo iptables -A INPUT -s n4 -j DROP
ubuntu@ip-172-31-36-51:~$ sudo iptables -A INPUT -s n5 -j DROP
ubuntu@ip-172-31-36-51:~$ sudo iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:6160
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:61023
DROP       all  --  n3                   anywhere
DROP       all  --  n3                   anywhere
DROP       all  --  n4                   anywhere
DROP       all  --  n4                   anywhere
DROP       all  --  n5                   anywhere
DROP       all  --  n5                   anywhere
```

Figure 2: The output from *iptables* on a node after partitioning

correct usage of Raft will lead to failures in our experiment even though the underlying implementation is correct.

Below is a brief description of each system we tested:

3.1.1 LogCabin

LogCabin [4] is a replicated consistent storage system written in C++ intended to store metadata. It is used as a metadata store for the RAMCloud project and uses Raft as a consensus protocol. The author of LogCabin, Diego Ongaro, is also the original author of Raft. We tested LogCabin with the default client implementation.

3.1.2 etcd

CoreOS is an operating system for large server deployments. To manage the configuration for these deployments, CoreOS stores data in etcd [6], a distributed key-value store. Much like LogCabin, the system is built to store small amounts of highly replicated data. etcd is written in Go and has a number of client libraries.

3.2 Testing Infrastructure

Timber tests consist of two independent components: A client program that writes to the cluster and a script that orchestrates network partitions.

In our tests, the client program writes a set of integers to an array stored in the Raft log. Since both Raft implementations expose a key-value store, we pick a path and write the set as the contents of the file. It is important that the writes from the client program can be easily verified. We therefore write a sequence of monotonically increasing integers to the set starting at 0. Once all writes are done, we should expect to see the exact same list in the Raft log without gaps, duplicates, or reordered values.

After completing the writes, the client program checks that numbers are present in the final set if and only if they were acked by the client. Let $A = [a_1, a_2, ..., a_n]$ be an array of acknowledged writes and let $R = [r_1, r_2, ..., r_m]$ be the contents of the file after all writes have completed. If the system handles network partitions correctly $A = R$. Otherwise, we can identify several types of failures: Survivors are writes that were persisted but not acked and are defined as $S = R - A$. Dropped writes are writes that were acked but not persisted and are defined as $D = A - R$. Note that R, S, and D are all mulitsets. Our client programs output the following values:

Delay/Part.	$\|A\|$	$\|R\|$	$\|S\|$	$\|D\|$	$\|S\|/\|A\|$	$\|D\|/\|A\|$
No/None	6091	6091	0	0	0.0	0.0
No/One	5358	5250	0	107	0.0	0.0202
No/Two	4394	4394	0	0	0.0	0.0
Yes/None	1610	1610	0	0	0.0	0.0
Yes/One	1280	1255	0	28	0.0	0.0219
Yes/Two	1041	811	0	230	0.0	0.221

Figure 3: Results from running Timber against etcd

1. The time it took to complete the client program
2. $|A|$: Number of acknowledged integers
3. $|R|$: Number of integers in the array
4. $|S|$: Number of present but unacknowledged integers
5. $|D|$: Number of acknowledged but not present integers
6. $|S|/|A|$: Percentage of survivors
7. $|D|/|A|$: Percentage of dropped writes

While the client program is running, a second script partitions and slows down the network. As described in 2.2, Timber provides a REPL to interact with the network. The REPL can also be given a file which will execute each line of the file sequentially. The file consists of a combination of *partition*, *snub_nodes*, *slow*, and *heal_cluster* commands, each of which are run against the cluster in parallel.

We ran the following scripts:

1. No partitions, full speed network
2. No partitions, slow network
3. Random single-node partitions, full speed network
4. Random single-node partitions, slow network
5. Random two-node partitions, full speed network
6. Random two-node partitions, slow network

The partitions are generated randomly for our script initially, but the tests on different implementations use the same script for easier comparison. We run *partition* and *slow* on the nodes in parallel to have the partitions happen as close as possible to instantaneously, thereby only creating symmetric link failures and slowdowns. The etcd client is configured with a one second HTTP timeout; the LogCabin client does not appear to have configurable timeouts, and operates over TCP directly rather than HTTP. Although the backend implementations also support timeouts, they are not configurable and we found the HTTP timeouts sufficient. The scripts all run for five minutes.

Delay/Part.	$\|A\|$	$\|R\|$	$\|S\|$	$\|D\|$	$\|S\|/\|A\|$	$\|D\|/\|A\|$
Yes/Two	1005	1009	0	4	0.0	0.0040

Figure 4: Results from running Timber against etcd without redirects

3.3 Results

3.3.1 etcd

Figure 3 shows the results of running Timber against etcd. There are a few interesting observations we can draw from the data. First, it is clear that network delays affect the throughput of the algorithm. The examples with regular network speeds process about four times as many elements as those with network delays. Additionally, partitions decrease the number of elements the protocol can process by 15-20% *per partition*. When the client programs partition the network, they never heal the cluster, so the entire script runs with some nodes partitioned.

Second, and perhaps more interestingly, etcd drops a significant number of writes. With one partition for both fast and slow networks, about 2% of writes are acked but not present in the result set. With two partitions and a slow network, 22% of writes get dropped. We would have expected the the fast network with two partitions to show dropped writes too, but we are unsure about why this does not happen. We plan to look into this in the future. Investigating the circumstances under which these writes get dropped, it appears that the dropped writes take place at a leader that gets partitioned out. It seems that A node that was partitioned in the previous partition then gets elected leader, although Raft should prevent this by requiring that the leader has a log that is sufficiently up to date. We plan on dissecting the exact conditions further in future work.

During our testing, we noticed that etcd by default allows redirects requests to follower nodes to their leader. While this, if implemented correctly, should not compromise the safety of the system, we found that disabling this behavior and only accepting writes at nodes who believed they were the leader improved the safety of the protocol. Figure 4 shows the results of running one test against a cluster under that configuration. We examined the logs for this example and found the output that can be seen in Figure 5. In this test, the value 92 was dropped. At the beginning of this part of the log output, nodes 3 and 5 are partitioned. We see that the value was acked on node 4 but that node 4 and 1 get partitioned shortly after. These two nodes must both have 92 in their log since our cluster has five nodes. Shortly after the end of the log output shown here, node 2 becomes the leader. Since 92 was dropped, this must mean that node 2 did not successfully accept the write but still acknowledged it or that node 4 acknowledged the write before it was applied at nodes 1 and 2. We are looking into the exact cause of this bug and plan on filing it with the etcd maintainers.

3.3.2 LogCabin

Figure 6 shows the results of running Timber against LogCabin. Since LogCabin was written by one of the authors of the Raft paper, one might expect an especially airtight implementation of the protocol, and indeed we did not observe a single missing or duplicated write in any of our tests.

```
1418285357 WRITE host=4 value=92
1418285357 ACK host=4 value=92
1418285357 PARTITIONED left=4,1, right=2,3,5,
1418285357 WRITE host=4 value=93
1418285357 PARTITIONED host=4 value=93
1418285357 WRITE host=5 value=93
1418285357 PARTITIONED host=5 value=93
```

Figure 5: Log output when running etcd without redirects

Delay/Part.	$\|A\|$	$\|R\|$	$\|S\|$	$\|D\|$	$\|S\|/\|A\|$	$\|D\|/\|A\|$
No/None	12294	12294	0	0	0.0	0.0
No/One	5306	5306	0	0	0.0	0.0
No/Two	584	584	0	0	0.0	0.0
Yes/None	2332	2332	0	0	0.0	0.0
Yes/One	1024	1024	0	0	0.0	0.0
Yes/Two	142	142	0	0	0.0	0.0

Figure 6: Results from running Timber against LogCabin

Partitioning nodes did, however, have a large effect on performance: with or without a network delay, partitioning one node reduced the write throughput by a factor of 2, and partitioning two nodes reduced throughput by a factor of 20 (relative to the base case), which is a much larger difference than we observed with etcd. We are not certain of the cause of this discrepancy, though we suspect it may have something to do with differences in heartbeat timeouts or the way leader elections are handled. We plan to investigate further in future work.

3.4 Challenges

The Raft implementations we test have had the advantage of time to handle edge cases and to specify their implementations to be optimized for use-cases. We have had to think out of the box on how to break their implementations. Raft is designed to be able to handle network failures and faulty nodes, and it was our task to experiment with Raft's limits – how many failures in conjunction could it take? Raft stipulates that it can handle up to f failures (where $2f+1 = N$), so we adhere to this; however, within this limitation, we have tried to determine a combination of failures that could break both LogCabin and etcd.

Another challenge is getting sufficient information to make the output useful. For now we have decided what to measure, and recording these measurements under ideal network conditions gives us a baseline to work against when dealing with a slow or partitioned network. While performance issues are apparent and relatively easy to diagnose, correctness issues are harder to trace. When Timber identifies errors in the protocol, it is still difficult to discover exactly what went wrong. We have diagnosed some issues here, but without additional logging as Timber writes to the cluster, we can point to issues but not their solution.

4. FUTURE STEPS

In the next version of this paper, we will have implemented and we will be able to discuss the following steps:

- We have observed *how* the Raft implementations break, but we have not yet explored *why*. Although a precise description of the steps that led to the failure is difficult without comprehensive logging, we can at least examine the state of the cluster before, during, and after the error. For example, it would be helpful to know which nodes are partitioned, which nodes were previously partitioned, which node is the leader, and which node the write occurred at. All of this information is available, but we have not had the chance to analyze it.

- On a similar note, it would be interesting to keep more detailed statistics on how how partitions affect the client. We only track the number of elements in the final array, but it is not clear if the number of elements decreases with the network delay because of internal delays in Raft or because of client timeouts. Knowing why the number of elements is smaller may help us discover how delays affect the Raft protocol.

- Currently, all partitions are symmetric and delays are applied to all nodes. There are, however, no guarantees that a system resilient to those failures will be resilient to asymmetric link failures or delays at only some nodes. We plan on extending our Timber API to support these types of failure modes.

- Our process is currently mostly automatic, but there are a few places in which we could automate our error detection. For example, a script to parse and cleanly display results to directly compare results under different network conditions would be valuable. This would make Timber easier to deploy on new systems and immediately derive value from.

5. REFERENCES

[1] J. Burke, R. Rygaard, and S. Stathatos. ”timber github”. *https://github.com/rasmusrygaard/timber*, December 12, 2014.

[2] K. Kingsbury. ”call me maybe: Carly rae jepsen and the perils of network partitions”. *http://aphyr.com/posts/281-call-me-maybe-carly-rae-jepsen-and-the-perils-of-network-partitions*, Accessed November, 2014, May 17 2013.

[3] K. Kingsbury. ”call me maybe: Postgres”. *http://aphyr.com/posts/282-call-me-maybe-postgres*, Accessed November, 2014, May 18 2013.

[4] D. Ongaro. ”logcabin github”. *https://github.com/logcabin/logcabin*, Accessed November, 2013.

[5] D. Ongaro and J. Ousterhout. ”in search of an understandable consensus clgorithm”. *Draft of October*, 7, 2013.

[6] Y. Qin. ”etcd github”. *https://github.com/coreos/etcd/tree/master/raft*, Accessed November, 2013.