RAFT Under Uncertainty - Simulating Lossy Network Connections

Leon Bi Stanford University leonbi@stanford.edu

Michael Chang Stanford University mchang6@stanford.edu

Thomas Jiang Stanford University twjiang@stanford.edu

Abstract

RAFT (Ongaro and Ousterhout, 2014) is a distributed protocol used to synchronize some data across a number of replicated servers. However, while the use of RAFT has been well documented for database purposes, storing key-value pairs containing integers or short strings, there has been little research into how RAFT handles replication of large files. In our paper we analyze the performance of the distributed algorithm RAFT under the condition of lossy connection between server and client such as what might be encountered over a mobile network. More specifically we investigate how RAFT implementations handle lost packets or dropped server-client connections when transmitting large files such as photos or video. This question is important as the increasing use of cloud photo storage, especially through mobile phone, necessitates the need for better replication protocols for photos and videos. We conduct simulations with the existing RAFT implementation, PySyncObj (Ozinov, 2017), under two modes (abort-if-dropped, and commit-if-dropped) with a variety of network and find that neither mode is satisfactory. Therefore we necessitate the need for a new protocol built on top of RAFT to deal with large files over lossy networks and suggest one such possible protocol.

1 Introduction

Raft was originally developed in an effort to create a more understandable consensus protocol over Paxos. It has since been implemented as the foundation of many large-scale distributed storage systems. Consensus means that multiple servers can agree on the same information across a network. The cluster of distributed servers interacts with a client system that can potentially send data over the network to be replicated within the Raft distributed system itself. The servers within the cluster can either be a follower, candidate, or leader. The servers periodically send heartbeats and over time will elect a singular leader. In Raft, only the elected leader should connect with the client and consequently all other nodes can only receive the data from the client once the leader has fully received the data from the client and then it sends it to its followers. The issue lies when the client wants to send an abnormally large file such as a photo or a video over a lossy network. The leader node would have to wait for the entire file to be uploaded from the client, which itself is a slow task under a lossy network, until it finally begins attempting to replicate the file to each of its peer nodes. Additionally, in the event that the leader node goes down in the middle of receiving the large data file from the client, the client would have to wait for a new leader to be elected and then begin the file upload again

from scratch.

2 Approach

To more efficiently send and replicate large media files, we propose a method of splitting the file data into fragments and reassembling the fragments back together at a later point in time once all fragments have been collected by the given server node. Before sending data, the client will send an initial request to inquire from the server which fragments for file path k have already been received (note that the file path k will correspond to the key for the corresponding data file in the key value store (KVS) on each server). This initial request will also include the total number of fragments expected for this one data file. The server will then respond with a response object that contains which fragment indices are missing for the complete data file.

If the file has never been sent before or no complete fragments have been successfully transferred, then the server will send a response back including all of the fragment indices. If some of the fragments have already been received, then the response will only include the remaining fragment indices.

Consequently, as soon as fragments are fully sent they will begin to be replicated among the raft cluster as per the original consensus protocol. As you can imagine, in the event that the network is partitioned while the client is sending data to the leader, fragments that have already been fully sent will be saved and replicated before the rest of the large data file fragments have been received.

Additionally, if the leader node experiences failure or a new leader is elected before all fragments have been sent, the client can reconnect to the new leader, which most likely already has fragments of the data file that the client is attempting to send, and continue sending the rest of the fragments that the new leader is missing. This prevents the client from restarting the file upload process again without the fragmenting.

It should be noted that our addition to the Raft protocol is reliant on the fact that the client is trustworthy and won't change the fragments of the data file halfway through file upload immediately following a network partition or leader change.

To reassemble the individual fragments and handle fragment uploads we implement an assembler class. The assembler tracks what fragments are complete, uploads completed fragments, and reassembles uploaded fragments when a client pulls data from a node.

3 Experiments

We ran three types of file upload experiments from the client to a Raft cluster of 6 server nodes for a file of size 325,811 bytes in order to simulate the impact of our proposal: (1) We simulated uploading a large media file with no packet loss. (2) We simulated uploading a large media file from the client to a Raft cluster with $x\%$ packet loss. (3) We simulated uploading a large media file splitting the file into 10 fragments from the client to a Raft cluster with $x\%$ packet loss.

For each of these experiments we added a

random chance (low: 0.1, medium: 0.3, and high: 0.5) for a network partition between the client and the server, meaning the client has to reestablish connection and restart its upload. Ultimately, we logged the round trip time to send and subsequently read the uploaded file.

The simulation was conducted with all nodes hosted locally on separate ports. We created an abstraction for packets (since python has few ways to directly interface with packets). We divided our 325,811 byte file using a packet size of 1000 bytes, for a total of 326 packets. Since over-the-wire communication locally happens almost instantaneously we impose a 60ms delay (an average RTT in a non-local simulation) server-side before responding to any received packets.

A single round consists of the client sending all remaining packets to the server it is connected to and waiting for the server to respond with the packets remaining before continuing to the next round. After every round there is a chance for a network partition, at which point the connected node loses all non-checkpointed packets. Although this implementation is imperfect (network partitions should be allowed to occur mid-round), with enough aggregated trials we can minimize the impact this imperfection has on our results.

4 Results

For all reported results, we ran each simulation of a given <packet loss, network partition> pair 50 times and averaged the results.

As a sanity check the first experiment we ran was for the original raft implementation with no packet loss to simulate a stable network (see fig 1). In the resulting graph you can see that the time it takes for the file to be uploaded and read again is independent of percentage chance for a network partition. This is a direct result of the imperfect round system described above as all uploads complete in a single round before any chance to network partition. We also see that in a stable environment our proposed implementation slightly underperforms which can be attributed to the overhead associated with uploading fragments as soon as they are complete.

Next, we ran experiments for Raft under a lossy network by simulating packet loss. We ran this for a RAFT cluster with and without our proposed fragmenting implementation for a low, medium, and high chance for network partitions between the client and the server (see fig 2, 3, and 4). In most experiments we saw that our proposed implementation decreased the round trip time for a large file to be uploaded and read again. In the simulation with the low chance of network partition (fig 1) we saw about the same performance no matter the packet drop chance until the packet drop chance was really high (> 0.4) where our proposed implementation is slightly faster. However, when the network partition chance and the packet drop chance are both relatively high like in figure 3 and figure 4, we can see that our proposed implementation does in fact decrease the round trip time a dramatic amount. In fact, with a 0.5 network partition chance and a 0.5 packet drop chance our improved implementation about halves the round trip time.

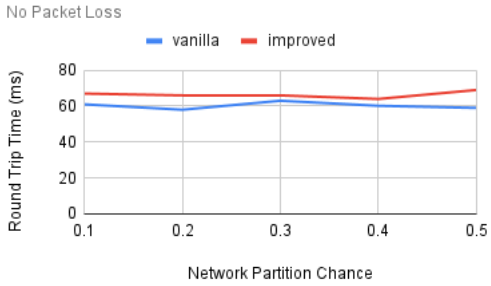

Figure 1

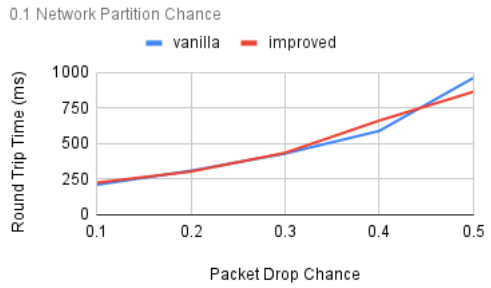

Figure 2

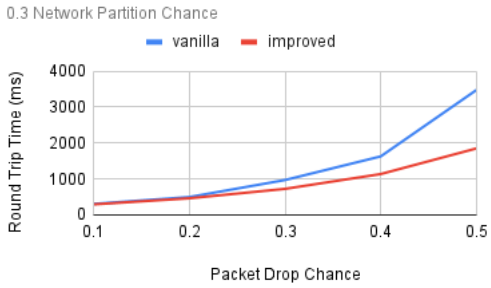

Figure 3

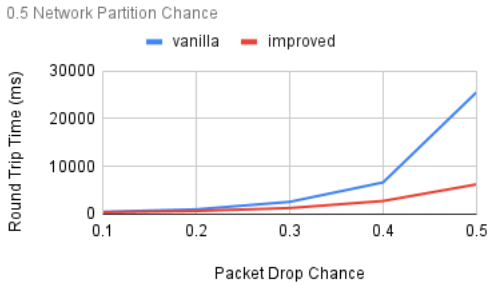

Figure 4

Packet Drop Chance	Newtork Partition Chance				
	0.1	0.2	0.3	0.4	0.5
0.1	210.0016891	276.0030114	303.0003068	358.206793	453.6029051
0.2	308.4039548	406.8086631	493.8016518	651.0090688	978.6021402
0.3	429.0067922	591.0037437	968.4099836	1412.403561	2554.208864
0.4	588.0035387	891.0091122	1627.201894	2973.601852	6627.601647
0.5	960.0024253	1636.207049	3477.606787	8043.602811	25527.00533

Figure 5: Vanilla Raft: File Upload Round Trip Time (ms)

Packet Drop Chance	Newtork Partition Chance				
	0.1	0.2	0.3	0.4	0.5
0.1	224.9089244	247.6521904	289.90899	320.4586992	370.5029261
0.2	302.9034325	358.8057743	458.904946	594.7593627	640.2525363
0.3	434.2068707	434.2068707	722.1512102	1031.55538	1252.554453
0.4	660.4049695	803.4035594	1134.907512	1471.603034	2730.655299
0.5	862.5511688	1169.355592	1849.253252	3215.555697	6217.909892

Figure 6: Proposed Modified Raft: File Upload Round Trip Time (ms)

5 Future Work

As we continue to work on this project we hope to conduct even more experiments to get a better grasp on how our proposed implementation affects Raft performance. One metric we hope to capture in the future is the time it takes for a complete file to be replicated across the entire cluster and how our proposed implementation affects this time during varying amounts of network partitioning and node failures. Since a client can continue uploading fragments to a new leader without re-sending already sent fragments, we expect to see that our proposed implementation will perform better than the standard Raft implementation.

Additionally we also plan to explore the ability to fragment large files during reads from the server cluster. In the event that the client loses connection to one server and reconnects to another, we hope that the abstraction of fragmenting the data will allow for faster read times because the client will not have to restart the file download.

It would also be interesting to see the im-

pact of varying additional variables such as file size and cluster size on our simulations. Furthermore, tracking other metrics such as latency, throughput, RAM usage, and CPU usage would be fascinating to analyze.

Our current implementation also only considers append only use-cases which works perfectly for media files because they most likely won't change. However, if the user wishes to modify or delete files we must consider how the fragments will be updated across the cluster.

Finally, we hope to compare how this idea of fragmenting large data files performs amongst other consensus protocols. Does Paxos see similar results or does fragmenting data make the performance worse?

6 Conclusion

Through our simulations of packet loss and network partitions within an application of Raft, we were able to see how splitting up large media files into fragments and piecing them back together can improve the speed at which files are uploaded and subsequently read from a RAFT cluster. There are still many more simulations to run especially when adjusting for different sizes of fragments, different file sizes, and different cluster sizes. We hope that our initial findings inspires further work into testing the limits of existing consensus protocols.

References

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA. USENIX Association.

Fillip Ozinov. 2017. Pysyncobj. https://github.com/bakwc/PySyncObj.