Raft Technical Report

Zhaowei Huang **Tianyi Wang**

1 Introduction

In this project, we implemented a distributed key-value store, using Raft as the consistency protocol. The key-value store implementation is a simple HashMap the programming language provides, hence in this report, we mainly discuss about the Raft implementation, and the configuration of the Raft parameters as well as the experiment environment.
A key-value store service should provide basic *put*/*get* interface. The *get* interface is a simple operation, the client can call *get* grpc on any available server and get the value of the key. However, Raft doesn't guarantee an immediate update on each server, so it's possible that the get can return an old value. But the final consistency is guaranteed, as far as the server is operational. For the *put* operation, the client can call *put* on any available server, but the put request will be redirected to the leader in the cluster, and the request would be constructed as an Log entry in Raft and be distributed by the leader to all servers in the cluster. The procedure should be the same as the procedure in Raft paper.

2 Report Basics

In this report, we implemented the key-value store by Java 1.8, with gradle as the package management tool. All servers and clients run on the AWS m5.large instance.

2.1 Configuration

The server configuration can be set by our configuration file. In this experiment, we set election timeout in the range from 1000 to 1500 milliseconds, and the heartbeat interval to be 18ms. Also, since we use "chaosmonkey" to simulate the network congestion and partition, we have a rpc_timeout parameter for each rpc call timeout configuration. That is, the rpc caller would consider the rpc receiver down if it doesn't receive a response within rpc_timeout time. Currently, we set it to be 3000ms. Another parameter that can be set is the maximum threads for each server. Each server would have a thread-pool initialized at the time the server started. We now configure it to 50 threads.

2.2 Test cases

Besides simple put/get tests, we have 3 general test cases in our client code. One is the network partition test by using the chaosmonkey client. Another two are the combined con-

current client operation/throughput/latency, which are used to test the system's throughput and latency. We also provide a client UI that enables clients to manually enter requests to servers. We have thoroughly tested these cases and no unexpected behaviors are observed. The detail of the test case would be discussed in the Experiment part.

2.3 Modification to Raft

We have some tiny modifications in the Raft implementation. This doesn't change the way Raft works, but simply makes our programming easy to run on AWS instances.
The first modification is the *matchIndex* variable in leader's volatile state. $MatchIndex$ is used to tell leader its consistency with other servers in the cluster. In the raft paper, it is reset to 0 at each re-election. However, we found reset it to -1 makes our implementation more convenient, since we can simply consider matchIndex[i] == -1 as a 0 match, and the leader can always send matchIndex[i]+1 entry to the follower.
We also initialize commitIndex and lasyApplied to -1, which can be used to indicate the indices in the log list that has been committed and applied.
Another is the votedFor variable in election. We simply set votedFor to -1 if this server doesn't vote for any candidate.
We also have a new variable called inFlightRequest, which is an array that records the current server's pending requests to each server. We have this variable since we always met the case when a server is down, the leader will run out of its thread pool since it is keep sending heartbeat to the failed client and doesn't finish until the timeout. For example, we have the timeout 3000ms, and the heartbeat 18ms, so when server A is down, the number of unfinished requests from leader to server A would be $\frac{3000}{18} = 166$, which is much larger than the size of the thread pool. If the threads resource is unavailable, the leader can't send any more requests to other servers in the cluster. Therefore, we have the inFlightRequest to record the number of in flight request from one server to another, and halting the request if it exceeds the quota.

2.4 Experiment Environment

For all of these tests, we have tested them in several experiment environments, including 5 servers on one instance, 9 servers on one instance, 15 servers on one instance and 15 servers on two instances. All environments are on AWS m5.large instance, each has 8G memory and 2 virtual CPU.

3 Experiment & Result

3.1 Throughput & Operation Latency

We have tested the throughput and operation latency of client requests in 4 different environment.

- 5 servers on one instance

- 9 servers on one instance
- 15 servers on one instance
- 15 servers on two instances

The first three environments are used to compare the performance between different number of servers, and the last two to test the performance difference between different number of instances.
Tests are executed in the same way. We have one client on an AWS instance to run 10 threads, and each thread tries to call put/get request 1,000 times on the cluster. Therefore, we have a total number of 10,000 rpc calls on the cluster. We record the number of succeed put/get requests, as well as their latency. For each request, the client would consider the request failed if it does not received any response from the server during the rpc timeout interval.

3.1.1 put

Figure 1 shows the put result of different number of servers on a single instance. As we can see, the performance of put requests varies with the number of servers. The largest y-axis value indicates the final number of requests finished. For the 5-server test, the cluster processed most number of requests, the 9 servers cluster processed less, and the 15 servers cluster least. It indicates that the number of dropped requests increases when servers scale up. It is what we expected, since all servers are on the same AWS instance, and each server has limited number of available threads. When the communication between servers are heavy, they would not be able to handle highly-concurrent requests from clients, so the 15 servers cluster would drop the most requests. The slope of the line indicates the number of put requests each cluster can server in every second. We found that the throughput are almost same for each cluster.

Figure 2 shows the put result of 15 servers on one instance and two instances, respectively. We can find that the 15 servers on two instances has a better throughput than the one on a single instance.

As for the operation latency, we can found it in Table 1. The latency increases with the number of servers. It's reasonable since when there are more nodes in Raft, it would take more time for the cluster to reach consistency. Also, the two instances case have better latency than one instance case, since we run the instance in the same region, and the communication latency should be small. However, when we create two instances, each instance has a half server load than before, so the system resources become more available for each server.

3.1.2 get

The get requests have similar comparing result as the put requests performance test. But it has much better throughput and operation latency than the put request, since we serve

server count	latency/ms
9 on one instance	211
5 on one instance	189
15 on one instance	257
15 on two instance	197

Table 1: average latency of put requests on each cluster

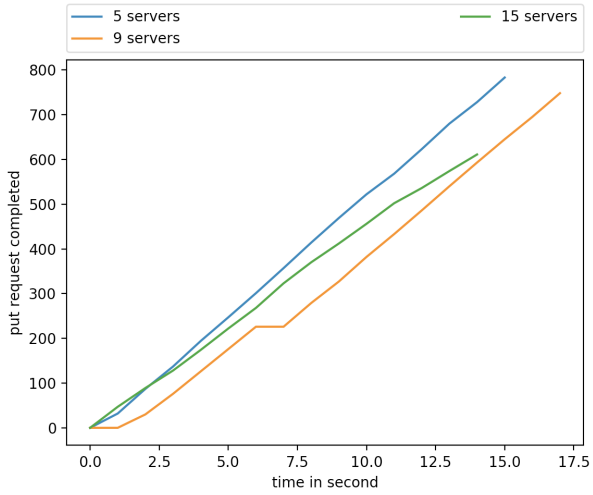

Figure 1: put request test on one instance

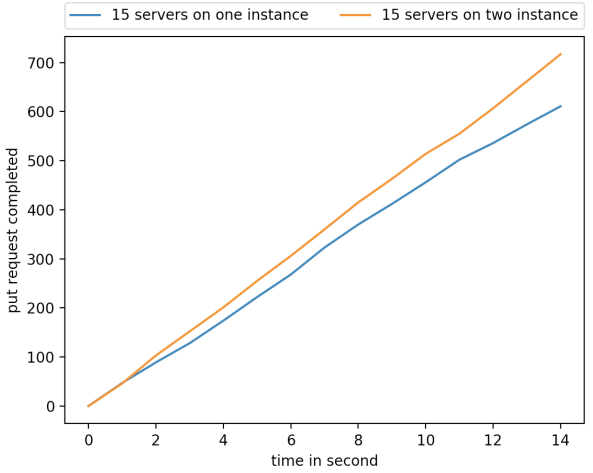

Figure 2: put request test on one/two instance

each get request on each server, without redirecting to the leader.

As we can see in Figure 3 and Figure 4, the cluster can serve much more get requests than serving put requests. We can also find the latency of get request on Table 2.

server count	latency/ms
9 on one instance	5
5 on one instance	15
15 on one instance	24
15 on two instance	9

Table 2: average latency of get requests on each cluster

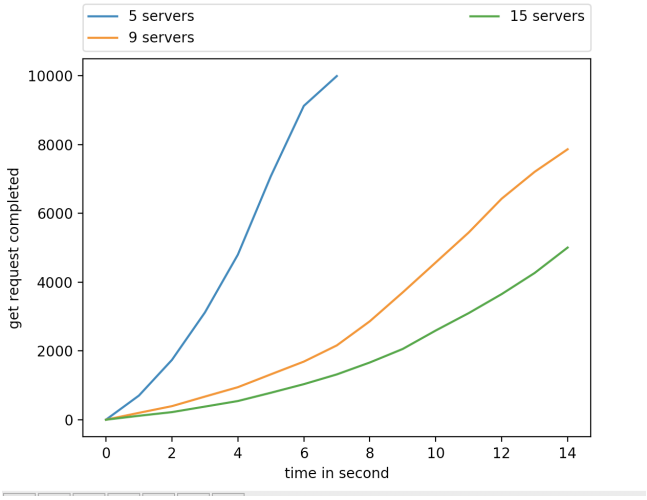

Figure 3: get request test on one instance

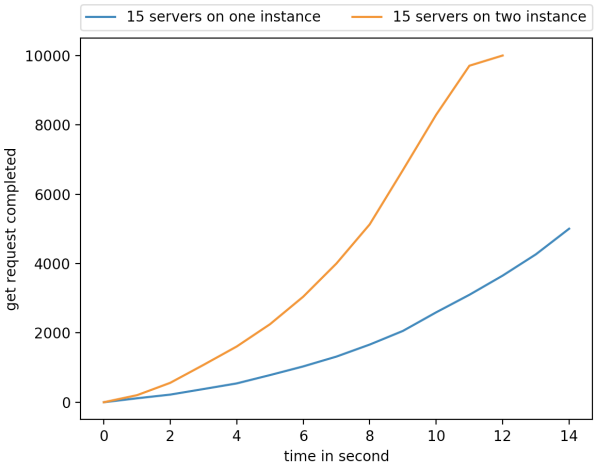

Figure 4: get request test on one/two instance

3.2 Different Key-Value Size

We test the throughput on one instance, with 5 servers running. The key-value size tested are 100, 1000, 10,000 and 100,000 bytes. The throughput graph can be found on Figure 5 and Figure 6. When the data size is small, the result is almost the same as the previous result on 5 servers cluster on one instance, where the key size is about 10 bytes. We think it's caused by the fact that 10 10000 bytes transmission can be considered as small package transmission, so its overhead is negligible when compared with the communication latency and raft cluster latency. However, when the key-value size is very large, which is 100,000 bytes, we can see an obvious reduce of throughput of put and get requests. Large packet or multiple packet fragmentation can obviously affect the throughput, but it only happens when it surpasses some threshold. We can also find the operation latency from Table 3 and Table 4, which has the same trend as the throughput.

key-value size	latency/ms
100 bytes	190
1,000 bytes	196
10,000 bytes	211
100,000 bytes	693

Table 3: average latency of put requests of 5 servers on one instance

key-value size	latency/ms
100 bytes	4
1,000 bytes	4
10,000 bytes	4
100,000 bytes	9

Table 4: average latency of get requests of 5 servers on one instance

3.3 Node Failure/Disconnections

There are two scenarios in Raft that is considered as a node failure. The first condition is when a node gets disconnected from all other servers, due to network congestion. We simulate this scenario by prompting chaosmonkey to send a matrix with 100 percent drop probability for all rpc calls from and to a specific server. A node failure can also happen when the server process abnormally exits, or the system shuts down. Since we run all virtual servers on one or two instances, we simulate this case by forcing a server process to quit. Both methods should produce the same output, such that as long as majority nodes work correctly, the entire system should eventually recognizes the node failure, and restarts a new leader election process if the failed node is the leader. Our focus in this section is to measure how the latency and throughput of client rpc calls are affected by these node failure. In our test environment, initially we have 9 servers running on the same instance. We randomly kill servers one by one until there are 5 servers alive. Notice that Raft protocol cannot converge

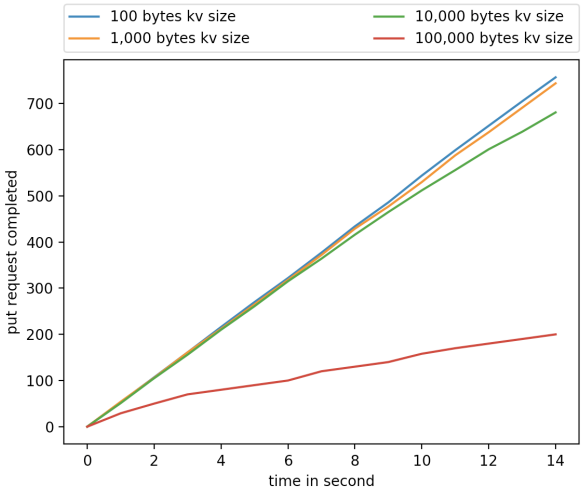

Figure 5: put request test on one instance

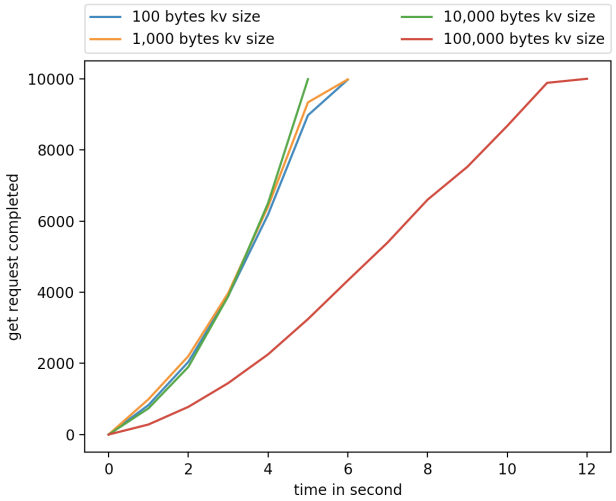

Figure 6: get request test on one instance

if the majority servers failed, so we set the maximum number of failed server to 4. Table 5 shows the put request latency and throughput in circumstances of different number of server failure. From the table, as the number of failed nodes increases, operation latency goes slightly down, and throughput goes slightly up. For each put request, the leader will firstly send append entries messages to all nodes, and upon receiving majority responses, the leader replies to the client. Since active nodes are still the majority and they can respond immediately, put request latency is not trapped by losing some follower nodes. However, as more nodes quit and therefore yield CPU, the remaining nodes have more system resources to explore, so every single leader append entry rpc call takes less time. Once again, our experiment shows that when some nodes fail, our hardware, 8 GB memory and 2 virtual CPU, becomes the major factor of client put request latency and throughput. The fewer

servers running on a single instance, the better performance should be observed.

failed servers count	latency (ms)	throughput (requests/second)
0	206	51.53
1	190	53.7
2	190	53.6
3	190	53.4
4	187	54.7

Table 5: put requests latency/throughput for 9 server on one instance under node failure

3.4 Reconnect

Once a leader node failed, it is significant that the entire system can elect a new leader and continue receiving client requests in a short amount of time. Fortunately, Raft ensures that if a follower node does not receive any heartbeat from the leader within an interval, that node will turn itself into a candidate and becomes a new leader if it receives majority votes. To measure the convergence time since the leader node fails, we use "chaosmonkey" to disconnect the leader node from the rest of servers, which simulates leader failure. Because we log most events with a corresponding time stamp, by referring to our logger, we can find the time difference between old leader node failure and the declaration of a new leader, Table 6 shows our result of 5 trials on 5 and 15 servers configurations, respectively. The first observation is that, for both server configurations, the re-connection time is under 1,500 milliseconds. Since our Raft implementation has election timeout in between 1,000 and 1,500 milliseconds, a new leader is always selected within one term, after the leader node fails and election timeout happens on other nodes. Meanwhile, the reconnect time is unaffected by the number of servers. When there are 5 servers, the system can recover from leader failure in average 1.17 seconds, while the recover time stays at 1.07 seconds when we have 15 servers running Raft. This phenomenon again suggests that network overhead can be negligible in our testing environment, since most vote requests can be delivered to all nodes instantaneously, no matter how many nodes are there. When the node recovers from system failure, like rebooting, our Raft implementation ensures that it retrieves all missing log entries by reading volatile states from the disk and learning from leader node append entry rpc calls.

number of servers	trial 1	trial 2	trial 3	trial 4	trial 5	average
5	1121	1372	1083	1144	1135	1170
15	1061	1030	1040	1120	1090	1070

Table 6: average reconnect time (in ms) of 5 servers and 15 servers after leader failed

4 Conclusions

In this project, we implemented the key-value store service based on Raft protocol. The performance in the experiment varies on the configuration of the experimental environment, like the number of physical machines, and the number of servers on each instance. There is also a huge performance difference between put and get request, since the put request has a much heavier overhead than put request. However, the throughput, as we can see, can not support a highly-concurrent put request workload, since it can only server about 50 put requests per second, when the environment is one or two instances. We ascribe this performance result to the fact that the small number of physical machine and the large number of servers on each machine. One of the major deviations from the Raft paper is that we set the random election timeout in the range between 1,000 and 1,500 milliseconds. By increasing the timeout, our implementation is able to keep system stable even when the network condition is unexpectedly bad, with up to 99 percent drop rate of rpc calls. Also, in the implementation, we use an exclusive lock to handle critical sections for each concurrent request, which would also cause a large overhead.