Implementation of Raft in Golang

Runze Xu, Xuhui Lu, Chuping Wang

May 2019

1 Abstract

We implemented a Key-Value store service using Raft consensus algorithm in Golang that is available, scalable and fault-tolerant. We evaluatd the correctness and performance of our system by utilizing Golang testing framework along with the ChaosMonkey service to create network partitions for the purpose of testing. This technical report is divided into three parts: environment, implementation, and evaluation.

2 Environment

2.1 Language: Golang

The system is implemented in Go language. Golang has built-in primitives, such as creating threads and condition variable, which make the multi-thread execution and inter-thread communication convenient to use.

2.2 AWS Configuration

We launched 5 m5.large instances on AWS. On each instance, we run 7 servers at most. Our testing have been done in three scenarios: one local machine, multiple connected local machines and multiple AWS servers. The first two scenarios are mostly for the purpose of debugging. For the evaluation of performance and correctness, we mainly ran testings on AWS servers.

3 Implementation

3.1 Configuration

Our default configuration parameters are as follows:

Heartbeat Interval	400ms
RPC Timeout	2000ms
Election Timeout Upper	1000ms
ELection Timeout Lower	500ms
Key/Value size	Total 32KB

3.2 Raft Service

To a great extent, the efficiency of Raft program depends on how it processes concurrent requests from both clients and peer servers while ensuring the parallel execution of the internal routines of the program. Thus, the lock is required almost everywhere. To best optimize the lock contention, we use the read-write-lock for data of server states. We never acquire a lock before an RPC call and release it after the RPC call returns because it may cause long period occupation of the lock and even deadlock.
There is no strict requirement about when the leader should send log entries to followers in the Raft. One strategy is to call the *appendEntriesToAllFollowers* function as soon as the server receives a put request from clients. The other is to wait until the next time to send a heartbeat, which is what we adopted. Moreover, we did the optimization as described in the paper that one *AppendEntries* call may contain multiple log records.
We also implemented the sequence number mechanism in the Key-Value service, which maintains the most recent request sequence number for each client's IP address. As described in the Raft paper, it performs request deduplication.

3.3 KV Service

The Key-Value service is in charge of the *PUT* and *GET* RPC calls. In addition, we implemented two more RPC calls for the purpose of testing: *ISLEADER* and *EXIT*.

ISLEADER(CheckLeaderRequest): Check the current server's membership. If it's leader, then returns true; Otherwise, it would return false along with the leader ID in the cluster.

EXIT(ExitRequest): To kill a process and exit the server instance. Unlike the *KILL* call in the ChaosMonkey, this RPC call could genuinely kill the process and exit.

3.4 ChaosMonkey Service

ChaosMonkey is designed to interrupt and test the correctness and performance of the Raft service. The Raft server would decide if to drop the incoming message based on the matrix entry's value modified by the ChaosMonkey client. Despite the basic *UPDATE* and *UPLOAD* gRPC calls, we implemented two more calls to guarantee the functionality: *PARTITION* and *KILL*.

PARTITION(PartitionINFO): Partition the cluster by sending partitioned server IDs to all servers. Eeach server would change its connectivity matrix based on if itself is in this partition.

KILL(ServerSTAT): Kill a node by changing the killed node's matrix value to let it drop any messages it receives to simulate killing a node.

To be noticed, our implementation of ChaosMonkey could issue *UPDATE* call 8 times per second.

4 Evaluation

With the help of Golang testing framework, We evaluated our implementation of Raft service in terms of its correctness and performance by running testing cases on clusters with different server number: 5, 11, 17, 23, and 31. Correctness test involves leader election, log replication, *PUT* as well as *GET* operation. Performance test focuses on concurrent operations, latency, and partitioned network performance. Deployment of the environment is via shell scripts, and auxiliary RPC calls like *START* and *TERMINATION* help to realize centralized cluster management and mock different scenarios. We also wrote helper functions like *checkLeader*, *generatePartitionParams*, and *resetMatrix* to boost our testing process.

4.1 Correctness

4.1.1 Leader Election

To evaluate the correctness of the leader election in Raft, we implemented various testing cases, involving continuously killing the leader and creating network partitions.

1. Testing_Killing_Leader

Decrease the number of servers from 31 to 16 by continuously killing the leader for each round of re-election. Verify if only one leader elected for each round.

2. Testing_Partition_Majority

Partition the network into one majority group and one minority group. Check the leader number in each group. If the leader was partitioned into the majority, the total leader number should be one; Otherwise, re-election should be completed in the majority and the total leader number should be two.

3. Testing_Partition_All_Minority

Partition the network into three groups with each size as two minor groups and another group with a single server. In this case, no re-election should be accomplished in any partition. The leader number should be zero in all partitions except the one contains the old leader.

4.1.2 Log Replication

To evaluate the correctness of the log replication, we simply do *PUT* operations and verify the json file of persisted log data of all servers.

1. Test_Log_Replication_1

PUT a randomly generated key-value pair to the cluster, and verify the log entry of each server by examining the persisted json file.

2. Test_Log_Replication_2

PUT a randomly generated key-value pair to the cluster, and edited the log entries of the majority to create wrong terms. *PUT* again and verify the log entry of each server by examining the persisted json file.

4.1.3 Client Operation

To evaluate the correctness of the client operation, i.e. *PUT* and *GET*, we need to generate the key-value pair and verify the result of *PUT* and *GET*. Because the *PUT* operation can be explicitly and implicitly verified in other test cases, we only wrote the basic testing script for the *GET* operation.

1. Test_Get_Operation

PUT a randomly generated key-value pair to the cluster, and verify the resulting value from issuing the *GET* operation.

4.2 Performance

4.2.1 Concurrency

To measure how many client operations our system can perform per second, we wrote a testing script which takes in two arguments. The first is the number of the concurrent clients and the second is the time duration clients will run. It launches the specified number of clients. These clients keeps doing *PUT* operations in a while loop and stop when the specified period has passed. We count the number of successful *PUT* operations in this period. With this script, we are trying to reach the threshold number of operations our system can handle in a given period of time.

In our experiments, we set the clients' running duration to be 5 seconds. The highest number of clients we have tested with is 1000. We observed and plotted the testing results on clusters with various server numbers. We can see from the figure that when the client number is smaller than 100, the number of successful operations grows linearly. It means we have not reached the threshold of our system. When client number exceeds 100, the curve turns sub-linear, which means we are approaching the threshold. It is spontaneous that the performance would decrease at some points when the client number increases to a certain amount. However, the peak for each cluster varies. For example, for the cluster with 5 servers, the peak happened at the client number of 750 was roughly 2000, but for the cluster with 23 servers, the peak was at the client number of 750 was approximately 1900, indicating the maximum performance differs for clusters of different sizes.

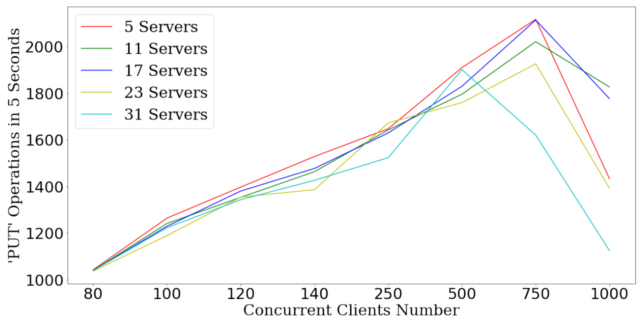

Figure 1: Concurrent *PUT* operations in 5 seconds

4.2.2 Latency

To measure the latency of client operations, we created a client and measured how many times it could put successfully to the cluster in 10 seconds. We repeated our experiments ten times, yielding stable results regardless of the server number. The times of successful *PUT* operation is 26 per 10 seconds, indicating the latency is approximately 384 milliseconds per *PUT* operation. As shown in Figure 2, the latency is not affected by the increase of running server number. It means our system is highly parallel when doing *AppendEntriesToAllFollowers* operation.

4.2.3 Node Failure

To simulate the scenario of node failure, we considered two situations: failure of the leader and failure of the follower. Since failure of the follower would not cause visible influence on the performance of the cluster, we chose to kill leader every time and observed what kind of effect it would have on the cluster. The result is the latency of each *PUT* operation after killing the leader.

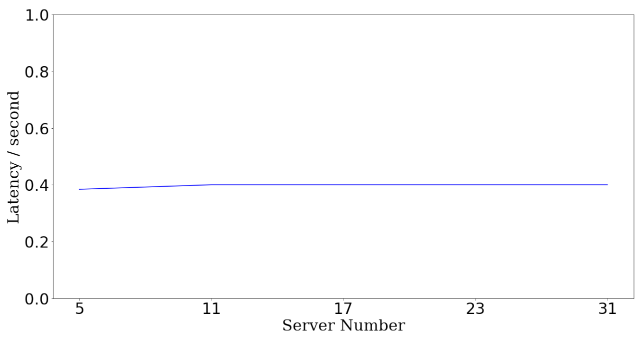

Figure 2: Latency of *PUT* operation on different clusters

Throughput: To test the throughput after failure of the leader, the same testing plan was adopted as before, simply by putting to the cluster concurrently using specified number of clients. From Figure 3, we could see there is a slight attenuation in the throughput. Such attenuation is caused by the re-election of the leader.

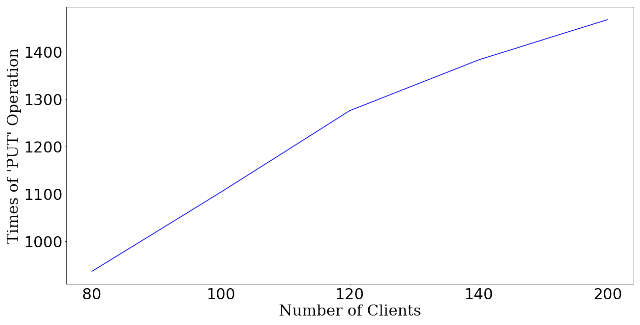

Figure 3: Throughput of *PUT* operations after leader failure

Latency: The average latency after the leader failure is 618.6 milliseconds. Compared with the 384 milliseconds in section 3.2.2, it is about 234.6 milliseconds slower. We suspect the reason is the cluster needs to select a new leader after the failure. Since the heartbeat interval is 400 milliseconds, and the timeout for the follower is 500 to 1000 milliseconds, such a decrease in the performance is reasonable and acceptable.

4.2.4 Key/Value Sizes

We evaluated the influence of the key/value sizes on the 31-server configuration. The sizes we chose are

1KB, 32KB, 128KB, 512KB, and 1MB.

Throughput: Since the size in testing is big, the overhead of data processing and communication increases accordingly. In this section of evaluation, only 10 clients were used. The variance in size has a significant influence on the throughput performance as shown in Figure 4. When the key/value size is smaller than 16KB, the system can still keep the level of throughput as shown in the previous section, hence the according data wasn't plotted in the figure. However, when the size exceeds 32KB, a sharp drop occurs. The system can hardly handle concurrency when the size is greater than 128KB. It can only finish 10 successful PUT operations in 5 seconds.

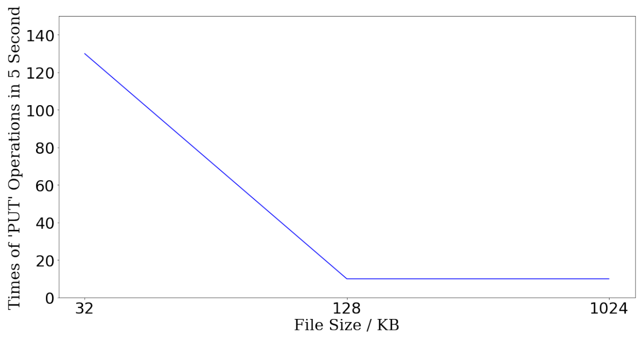

Figure 4: Concurrency of PUT on different key/value sizes

Latency: We plotted the size influence on latency as Figure 5 shows. When the Key/Value size is smaller than 128KB, the latency are at the same level, which is the lowest latency of our system. As the size grows to 512KB, the latency suddenly increases. When the size grows greater than 1MB, it is hardly to complete an successful operation in 5 seconds so the service becomes unstable. Therefore, we think 1MB is the upper bound for the Key/Value size of our system.

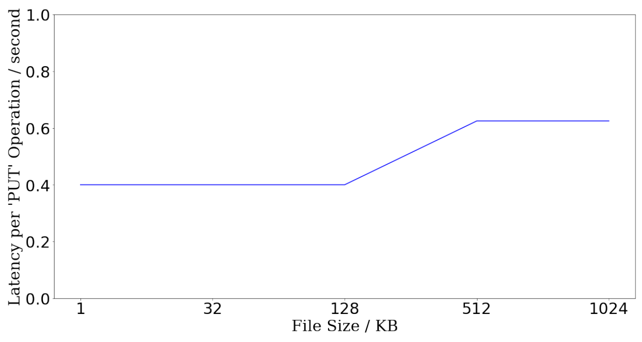

Figure 5: Latency of PUT on different key-value sizes

4.2.5 Reconnection

To measure how long it will take for clients to get service again after the failure, we designed two major test cases for evaluation. Both of them use the Chaos-Monkey service to form the specific network partition scenario.

1. Cutting off the leader node

Because cutting off one follower will not affect the functionality of the system, we chose to only cut off the leader in this experiment. After partitioning the leader alone, we tested how long it would take the client to acquire the service again, i.e., perform the PUT operation successfully to the remaining cluster.

2. Partitioning the network to all minor groups

If there exists a majority group in the network, the system would be still available. So we created multiple minor partitions to ensure there would be no leader elected after the partition occurs, and measured when the client could perform the PUT operation successfully again when the partitions are removed.
We ran the two test cases on 5, 11, 17, 23, and 31 servers respectively, and plotted the result as figure 6 and figure 7 show. From the figure it could be seen that, as the server number increases, the average time of performing one PUT operation basically stays at the same level with little variance. Thus, we could say the service scales when encountering the partition based on our testing results.

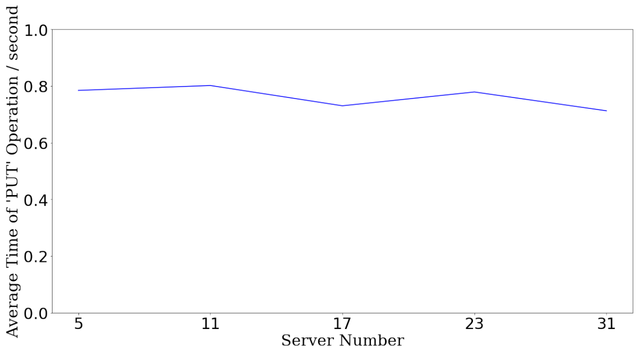

Figure 6: Latency of PUT after partitioning the leader

4.2.6 Timeout Effect

In this section, we evaluate the influence of the length of the election timeout on leader re-election time. Unlike what has been done in the Raft paper of plotting the cumulative percentage of leader re-election time, we tested the leader re-election time on 5 different election timeout choices. We fixed the lower bound

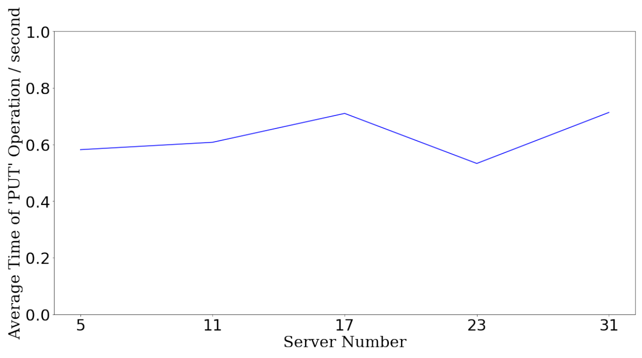

Figure 7: Latency of *PUT* after partitioning the network into minor groups

of election timeout to be 500ms and change the upper bound ranging in 500ms, 525ms, 550ms, 575ms, and 600ms. The experiment was conducted on the 31-server configuration. For each timeout length, we first partitioned the cluster into 31 single partitions, so that each server was isolated. Then, after waiting for all servers to time out (except the leader which never times out), we measure the time it takes for the client to complete the first *PUT* operation. The whole process was repeated 50 times for each timeout length, and then computed the average. The result is shown in Figure 8. In this figure, although the re-election time

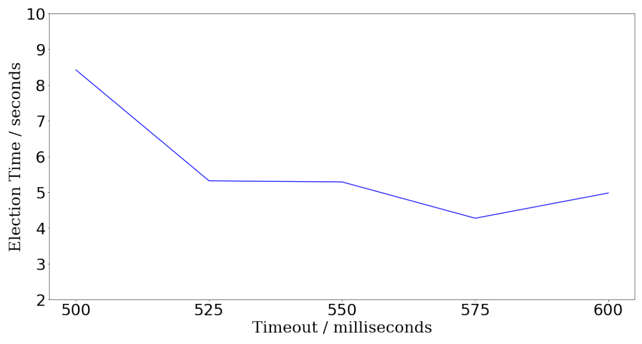

Figure 8: Influence of timeout on the leader election

of 600ms is higher than the 575ms. It is still obvious to see the trend that the narrower the gap between lower and upper bound of the election timeout is, the longer it takes to re-elect a leader.

4.2.7 Re-election in Extreme Cases

With the number of alive nodes decreases, it is supposed to be more difficult for the cluster to elect a leader. Based on this intuition, we would like to test how fast it is able to elect a new leader when the number of nodes varies.
We implemented a testing script to kill the emerging new leader over and over again. Every time a leader is killed, we began to detect whether there was new leader elected. The result in Figure 9 shows that the re-election time is very short and no major fluctuation occurs. It is because we combined the local Unix nanosecond-granularity timestamp along with the server ID as the seed to generate random timeout number.
However, we also witnessed a sharp increase when the number of node is 16. Suppose we have *2f+1* leaders, the result shown in Figure 9 indicates that the leader could be rapidly elected when *f-1* servers are killed. However, when *f* servers are killed, the leader election time sharply increased from around one second to about 2 minutes. We attribute this phenomenon to the Pigeonhole principle. To explain it, we can imagine the election timeout range (500ms to 1000ms in our case) is divided into *f+1* equally intervals. The interval in this experiment is about 31ms long, which is a short period of time that is not enough for a single candidate to collect votes from other servers. Since we have *f+1* servers, according to the Pigeonhole principle, it is highly possible that 2 servers will time out during the same interval. One of them will not vote for another. Thus, the leader would not be elected.
It is still confusing for us that the leader election time is extremely low when there are *f+2* leaders. It might simply because that the existence of one more server can relieve the influence of Pigeonhole principle to a large extent.

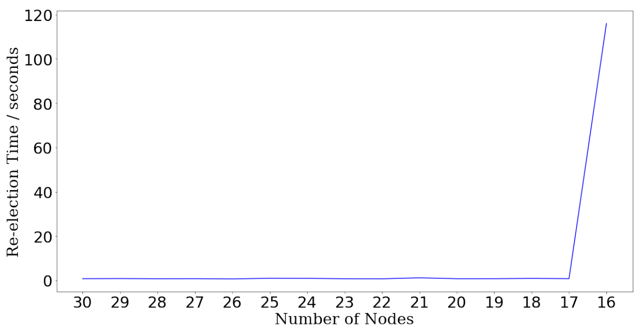

Figure 9: The effect of number of alive nodes on the election time

5 Future Work

5.1 Persistent Storage

In this project, we persisted the variables of Raft in the format of normal json file. It is convenient and pellucid. However, when trying to test the *PUT* operation of big files, we found it is not so extensible since the write/read operations of the large data would slow down our service.
There are several alternatives for this problem. First, we could use the file point to append proper contents

to the end of the file to avoid lots of I/O operations. More generally, using the database is a better solution. If we could stored our persistent data as tables in the database, it would reduce the extra I/O overhead of our service.

6 Conclusion

We implemented a Key-Value store system using Raft consensus algorithm. We designed test cases and conducted unit test both on local machines and AWS servers. The correctness of our system is guaranteed. We did our tests mostly on the 31-server configuration. The system could keep running for a very long time and is always available. To evaluate the system's performance, we first conducted all experiments listed in the checkpoint 3 handout. Then, we designed two other performance testing cases, namely the Effect of Timeout Length as shown in section 4.2.6 and Reelection in Extreme Cases as shown in section 4.2.7. The performance result illustrates both upsides and downsides of our implementation. Our system is scalable as the number of servers grows, but when the Key/Value size is big, it cannot provide the similar performance as the small Key/Value sizes.