RAFT Consensus Protocol Implementation and Measurement

Chih-hung Cheng, Xi Pu *University of California, San Diego*
c1cheng@ucsd.edu x3pu@ucsd.edu

***Abstract*— In this project, we create a system that implements the RAFT Consensus Algorithm to provide a basic key-value storage service to clients. The system is implemented with python and gRPC for both the server and the client, along with a Chaos Monkey feature to simulate various network failures. We set up our experiment locally and on AWS EC2 with different number of servers. We test the system performance including throughput, latency, crash recovery, etc. To verify the correctness of the system, we design various edge cases and cases under problematic environment. The project is on github.**[1] [2]

I. INTRODUCTION

This report concentrates on our implementation of RAFT, its experiment and analysis. Our implementation of RAFT server has leader/follower/candidate threads which are controlled by Condition Variable to ensure only 1 runs at a time. The leader/follower thread is responsible for creating its own thread for each vote/appendEntries request(and receiving corresponding response). However, our implementation is in python and doesn't include asynchronous I/O, both factors might have impact on our server performance for multithreading.

Our implementation is proven safe, available and scalable. Safety is proven by checking all the logs generated by our servers with concurrent client put; availability is proven by disabling up to n out of 2n+1 servers and ensure our distributed systems still function; Scalability is proven from 5 to 31 servers as in registerClient performance and put/get throughput performance (such as Put throughput vs number of servers). Because leader needs to send appendEntries(or heartbeat) to all other servers before replying to client, around the same latency with more servers means scalability. However, we found bottleneck exists at the leader due to its 7MB/sec network speed. Also we expect there should be degraded performance for number of servers reaches above 31 due to our synchronous I/O implementation and Python multithreading performance.

II. IMPLEMENTATION

A. Data structure

- Persistent states, all persistent states are updated synchronously
  - votedFor and currentTerm are saved and loaded in json format due to their small size
  - log[] is implemented with a list, which is saved in pickle (Python object serialization) since log[] might have a large size where its writing/reading performance is a concern
- Volatile states, commitIndex and lastApplied are just single variables
- Volatile states on all servers, nextIndex[] and matchIndex[] are both saved as Python lists with its index equal to server index

B. Leader Election

- Three threads representing leader/follower/candidate roles, which are never killed after creation during initialization of the server.
- These three threads are controlled via corresponding Condition Variables.
  - wait() and notify() are called during state transition.
  - waitFor() is used for pausing a thread for as long as election timeout, thus preventing the use of spinlock for better performance. Upon the elapse of timeout, we check again if state transition, more waiting or restarting the loop inside the function is desired.
- Candidate thread will request votes from each other server; Each request, including request and receiving response will be using a single synchronous thread.
- A leader/candidate will convert to follower whenever a request or response contains a term bigger than its current term, this is implemented as a separate function which is checked at either of the following 3 conditions:
  - Dealing with election vote request
  - Dealing with election vote response
  - Dealing with append entries response
- Encountering higher term while dealing with append entries will simply update its last update time.

C. Log Replication, appendEntries RPC

- Leader thread will send append entry request to each other server, one thread per request which requests and receives response synchronously.
- Applying to the state machine is done asynchronously with an additional thread, whereas upon finishing, it would notify threads that require state machine update such as client serverQuery RPC.

D. Client Interaction

- Client is always redirected to leader by the server it communicates to, the server gives its known leaderID as a hint.

[1] https://github.com/GateStainer/raft-server-python
[2] https://github.com/GateStainer/raft-client-cse223bsp19

- Client registers itself at the leader using registerClient RPC, where the leader would append register command to the log, and commit it; afterwards, it would reply OK.
- Put is implemented using clientRequest RPC, which carries clientID, sequenceNum and command, where clientID is used to check whether a client has registered, sequenceNum is needed to ensure its not an old request.
- Get is implemented using clientQuery RPC, which would wait until new round of appendEntries (heartbeat) to have finished, then query the state machine to find the result.

E. Chaos Monkey

- The Chaos Monkey matrix is uploaded/updated via RPC, and its done naively by having our clients send matrices to all servers sequentially.
- The blocking by Chaos Monkey is always implemented at the receiver side, it blocks when either a server receiving requestVote/appendEntries request, or server receiving response for either request.

We generate a random float between 0 and 1, compare with the matrix value, if Chaos Monkey blocks, it would generate a message in logging without any further execution.

F. Follower/candidate crash and recovery

It is implemented by first using Chaos Monkey to block all incoming/ outgoing traffic for the crashed server, wipe out all its RAFT related data in the volatile memory, then the crashed server will load the parameter from its disk and reconnect.

G. Debugging

Python logging is used, which have 2 corresponding handlers to print out and save to disk for our debug logging: time detailed to milliseconds, corresponding debug logs and variables.

H. Implementation that differs from RAFT or not stated in RAFT paper

- After a server grants vote to another server, its election timeout countdown will restart. The reason is that servers initiated as followers; thus if it grants votes and the countdown with election timeout. continues, it might timed out first and initiate voting before getting the append entries from the leader, thus leading to longer stabilization time.
- A local variable consisting of currentTerm is created before a candidate request vote, or a leader send append entries request; the currentTerm is updated upon receiving the response. The justification is that even though we use multi-thread for requests, we might have received the response which carries a higher term during sending request, causing the currentTerm to increment multiple times within a single election. Such occurrences prevented stabilization.
- Application of log[] to state machine is done asynchronously since it shouldnt be in critical path during log replication, it would notify. registerClient/Put RPC to continue and send reply since leaders require more than half of the servers to have applied the logs to reply OK.
- When a candidate receive appendEntries request, it compares the request term with its last log term (instead of candidates currentTerm); if request term is equal or higher, it will convert to follower. The justification is that sometimes a leader hasnt been established to append the no-op entry, a candidate might increment its currentTerm to some arbitrary high number due to bad network connection. And such term isn't meaningful but only affects leader election stabilization.

I. Technical limitations and improvement proposals

- Under condition of concurrent client operations, performance might be compromised, since our server implements synchronous I/O. Thus it could be improved as follows:
  - Async I/O should be implemented for better multithreading performance.
  - The RAFT could be implemented in languages which carries better multithread support.
- Upon failure to send appendEntries, leader would decrement the nextIndex by 1, however when the follower lacks lots of entries, such decrement would take a long time to the correct index. We would propose appendEntries should carry last log index in its response, thus the leader could refer to its value for nextIndex.
- When we send appendEntries, if the log entries in the request is too large, it should be divided and sent separately.

III. Experiment setup and evaluation

We develop our protocol locally for easy modification and testing. Final testing is performed on Amazon EC2 using t2.micro. We are running only 1 process/server on each VM because t2.micro only has one core CPU and for easier management. We scale the number of VMs from 5 to 31 to observe the performance trend. Client testing is performed locally for easier access and modification. We are testing the performance about leader election, log replication, operation throughput, operation latency and concurrent client.

A. Time for leader election

We experiment to find leader election time when old leader crashed. The manipulated variables are server numbers and max election timeout. We measure the time for a candidate to become follower or a leader. We observe that:

- Election time result slightly decreases when server number increases. Since with more servers, its more likely that a server has election timeout closer to half of the max election timeout, and with 300ms election timeout, all of our testing elects a leader during the first round of election. However, we expects that with more than 31 servers, the chance for having election collision would increase, thus election time result would increase.

- Further, the election result time increases linearly to max election timeout. It is reasonable because only after election timeout passes that a new election round may start.

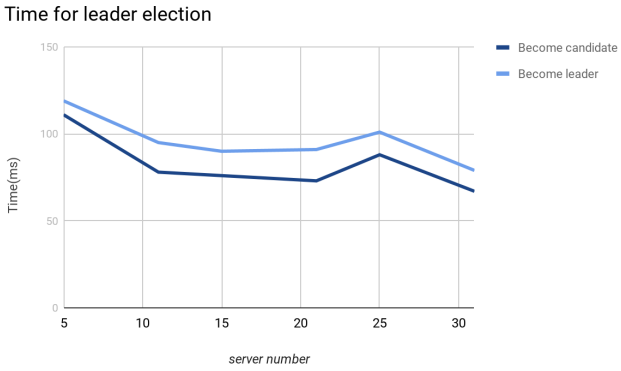

Fig. 1. Leader election vs server number

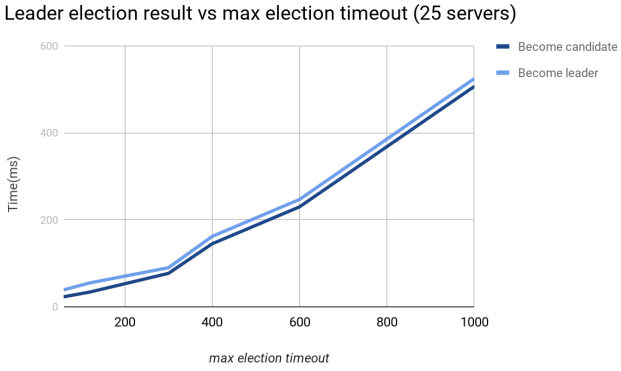

Fig. 2. Leader election vs max election timeout

B. Time for log replication

We experiment to find log replication time, which is the time it takes to have the log replicated on more than half of the servers. We set key/value size to 1 KB. We range over different numbers of servers to see its influence on log replication time. We see that log replication time increases sub-linearly with number of servers. Even though the result isn't flat as expectation, we argue that the the performance is due to the bottleneck at the leader, which is around 7mb/sec since log replication time only increases after we reach around 13-15 servers and the total network speed at the leader is shown on Amazon to be around 7mb/sec. Another subtle trend we can spot is that after 25 servers, the performance decreases even further by around 5%, the possible reason is either small experiment data set or that CPython doesn't scale well with threads for CPU-bound tasks.

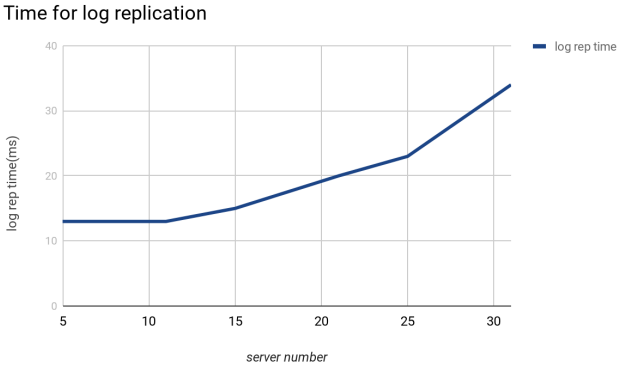

Fig. 3. Log replication time vs server number

C. Operation throughput

We test the throughput for different kinds of operations, including client registering, put, get, uploading and updating Chaos Monkey. The independent variables include server numbers, key/value size. We change them and check the performance on operation throughput. Also, we test whether the number of crashed servers has impact on operation throughput.

The throughput for registerClient remains almost flat with slight decrease when number of servers increases, same as expectation. It doesn't experiment the bottleneck since each registerClient only adds a single log entry and thus only limited by the round trip time of our gRPC request.

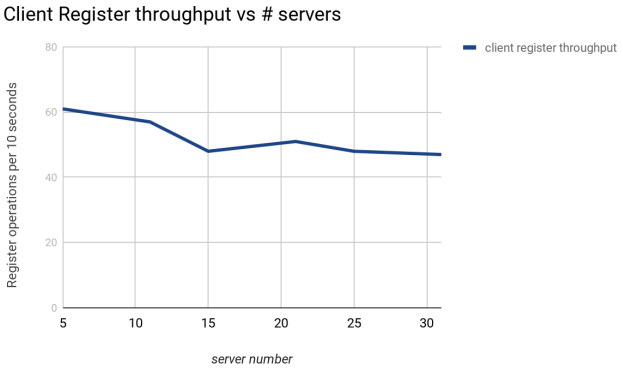

Fig. 4. Client register throughput vs server number

The throughput for put(clientRequest RPC) remains almost flat with slight decrease when number of servers increases, same as expectation. It doesn't have bottleneck since each put only adds a single log entry and thus is only limited by the round trip time of our gRPC request.

The throughput for put(clientRequest RPC) remains almost flat with slight decrease when up to n servers out of total 2n+1 servers crash, same as expectation.

The throughput for get (clientQuery RPC) is inversely proportional to key/value size. Since get (clientQuery RPC) is

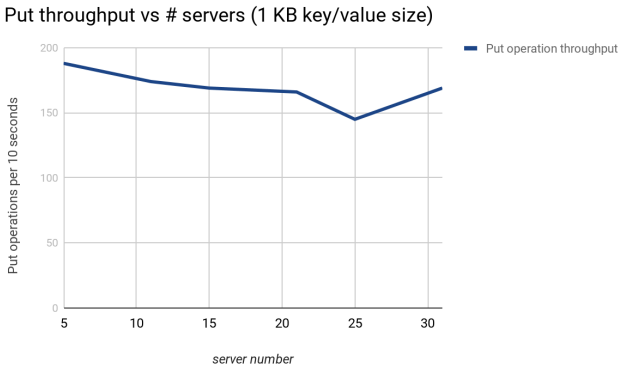

Fig. 5. Put throughput vs server number

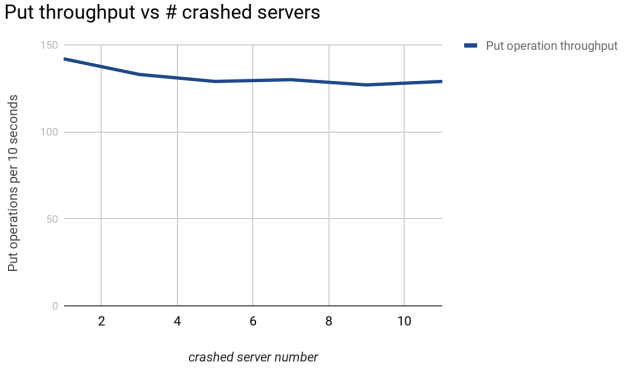

Fig. 6. Put throughput vs crashed server number

implemented by having leader send appendEntries(heartbeat) and then return the value for its state machine, the primary bottleneck of get is thus the leader network speed, which is limited to 7mb/sec.

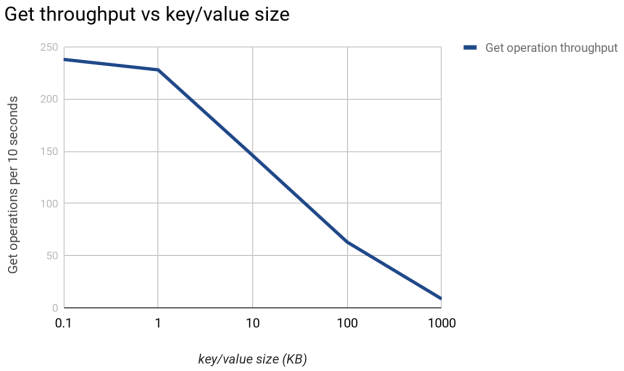

Fig. 7. Get throughput vs key/value size

The throughput for put is almost inversely proportional to key/value size, same as expectation. The leader is responsible for send appendEntries for the put key/value set to all servers and only apply to state machine and reply upon more than half of servers receiving the entry. Since there is a bottleneck at the leader of around 7mb/sec, key/value size past 1MB will eventually requires change on election timeout since sending sending requests would take longer; key/value size past 10MB isn't event supported.

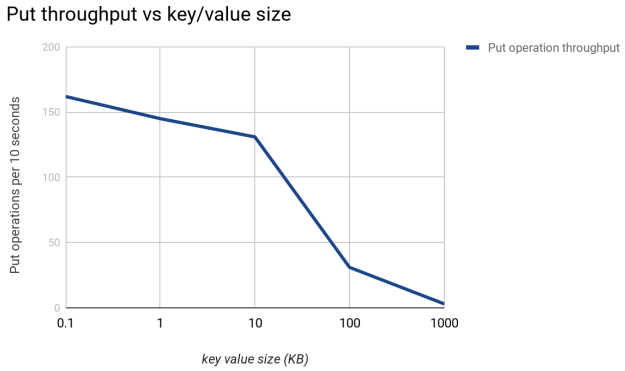

Fig. 8. Put throughput vs key/value size

We upload and update the Chaos Monkey Matrix in a synchronous way. So the throughput is almost inversely proportional to number of servers. We may consider changing it to asynchronous way and we assume that the throughput has little relation with server numbers.

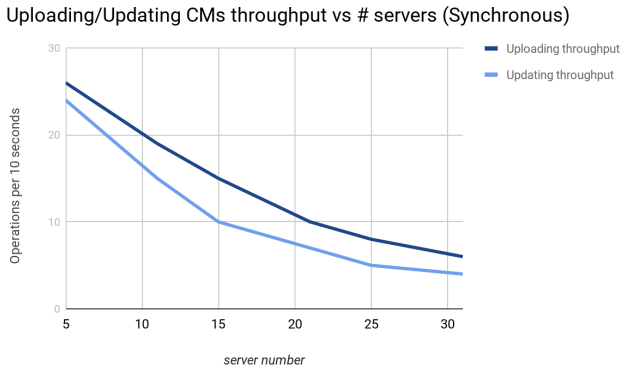

Fig. 9. Uploading/Updating CMs throughput vs server number (Synchronous)

D. Operation latency

Client registration throughput proved our implementation scalable from 5 to 31 servers. Since leader appends registration to the log and requires the leader to apply the log before replying OK, (where the application on leader requires half of servers to have committed the entries), thus the latency staying the same implies our implementation is scalable.

Put (clientRequest RPC) latency increases linearly with number of servers. The reason is that:

- There is 7MB/sec network speed limits at the leader
- There is overhead for creating and get multi-threads running

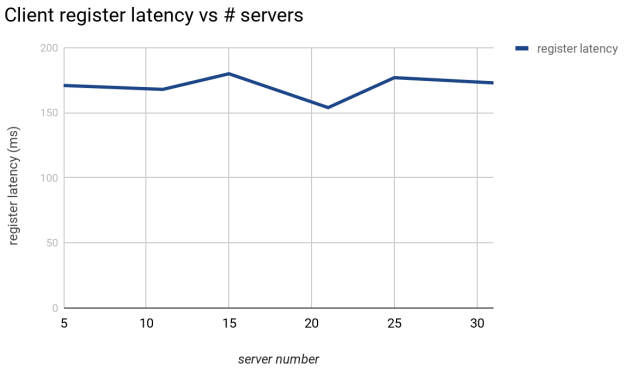

Fig. 10. Client register latency vs server number

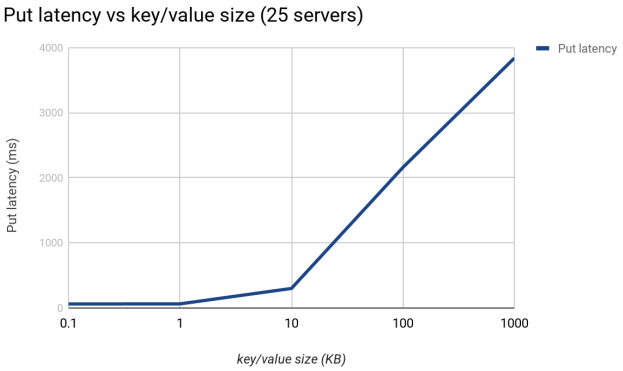

Fig. 12. Put operation latency vs key/value size

Both factors prevent the leader to send appendEntries instantly.

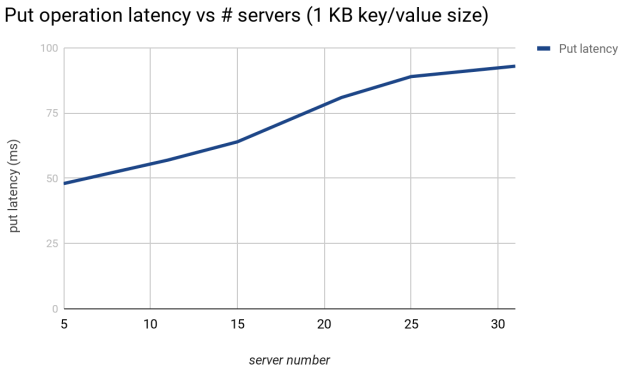

Fig. 11. Put operation latency vs server number

Put (clientRequest RPC) latency increase linearly with key/value size. Since there is bottleneck at the leader side to send out appendEntries, put latency increases linearly with key/value size. Note we have to adjust the election timeout here because, it takes longer than max election timeout to send large logs.

Put latency remains flat with number of crashed servers, same as expectation.

Get (clientQuery RPC) while remains flat first, and increases linearly with key/value size, which can be expected. It appears that leader network speed only becomes the bottleneck when key/value size is larger than 10kB.

Uploading/updating ChaosMonkey time increases linearly to number of servers, which is understandable since we use a for loop to update ChaosMonkey matrix on each server one by another.

E. Concurrent client

Put latency is linear to number of concurrent clients. We fix server number to 25 and key/value size to 1 KB with increasing number of concurrent clients The leader becomes the bottleneck when clients execute put operations

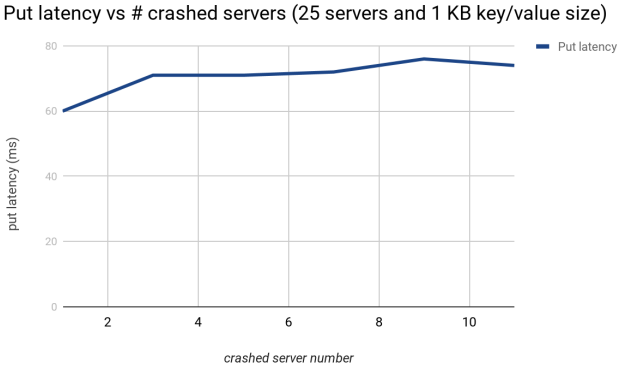

Fig. 13. Put operation latency vs crashed server number

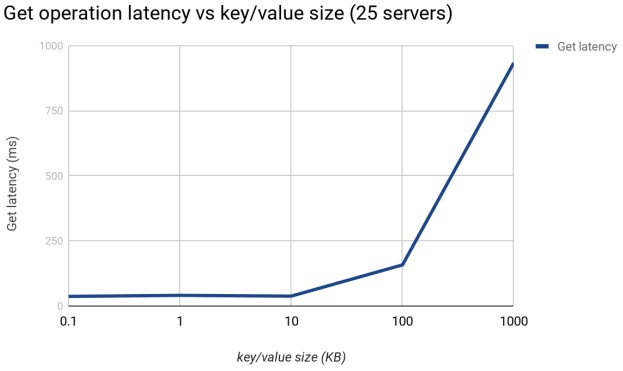

Fig. 14. Get Operation latency vs key/value size

concurrently. And the lock makes put operation wait for others finishing. As for concurrent put throughput, the total throughput increases linearly with number of clients when concurrent client number is fewer than 100. The distributed system can handle all clients put operations in a timely manner with limited number of clients. Our system can support a maximum concurrent clients of 160. If the number of concurrent clients exceed 160, there will be severe latency.

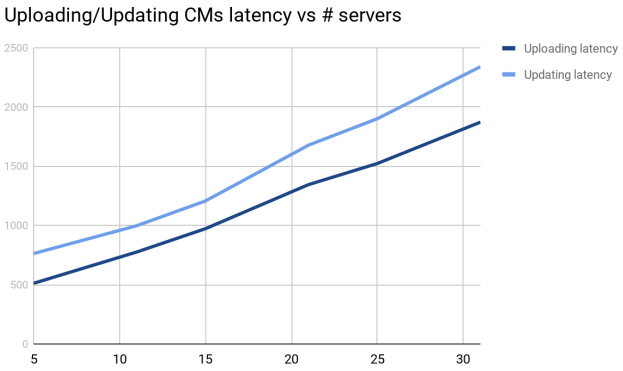

Fig. 15. Uploading/Updating CMs latency vs server number

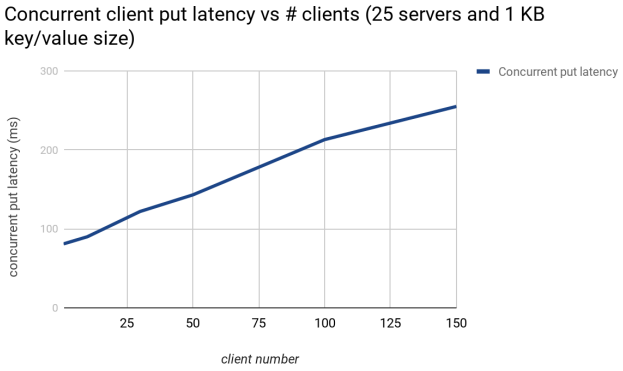

Fig. 16. Concurrent client put latency vs client number

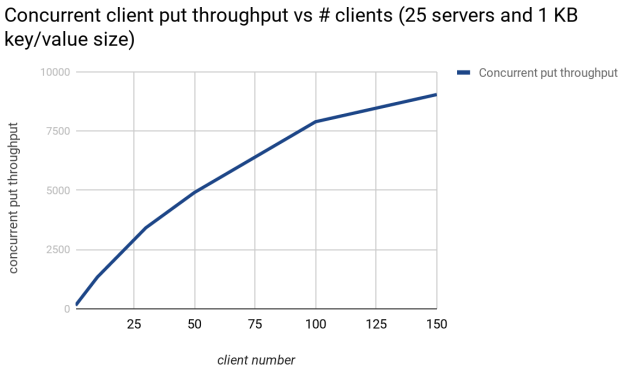

Fig. 17. Concurrent client put throughput vs client number

IV. Recovery Time

Recovery time increases linearly with log length. We test the recovery time for single server and servers. We fix server number to 25 and key/value size to 1 KB. We stop part of the servers and execute put operation for given times, then restart the crashed servers. To recover log entries on crashed servers, the leader needs to send AppendEntries gRPC to the server. The times leader sends AE is equal to missing log length and the leader needs to send all missing log entries in the final AE gRPC. We find out the bottleneck of recovery is the repeated AE gRPC sent by leader.

We spotted that current implementation would include the whole missing log no matter how long it is. We recommend that appendEntries should send missing logs separately when its too large.

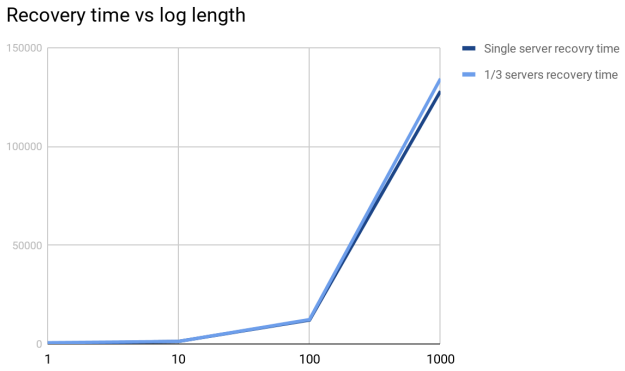

Fig. 18. Recovery time vs log length

V. Correctness

- No more than one leader at the same time. During election, only 1 leader will win the election
  - A candidate increments its current term, requests votes and it requires majority of the servers to grant votes in order to become leader.
  - upon becoming leader, in order to establish its authority, the leader would append no-op entry to log and send out append entries.

  While there is already a leader.
  - A server will maintain its follower state as long as valid RPCs are received.
  - A candidate will return to follower state when it receives a appendentry (heartbeat) with request term equal to or higher than its current term.
- RAFT can work with more than half servers working
  - Clients always redirect to leaders, and all the entries are maintained with both log index and term number.
  - A log entry need to be replicated on more than half of the servers so that the leader can change its leaderCommit index to the entry index.
  - A log entry can only be applied to the state machine when it has been replicated on more than half of the servers, and only upon completion it can reply success to the client.

VI. Conclusion

Our implementation of RAFT server achieved satisfactory result. Each of our experiment was carried for 5 times to acquire confidence level.

In terms of safety, followers ensures correctness by checking and deleting any conflicting logs before writing to disk. We had concurrent clients putting to the RAFT server and verified our result by taking the logs from all servers and run a check-up script to ensure their consistency.

In terms of availability, we imitate disabling up to n out of 2n+1 servers, and latency of our RAFT implementation remains flat with slight decrease when the number of crash servers increases.

In terms of scalability/performance, the performance of our RAFT implementation scales with number of servers either flat or sub-linearly in most test cases. Exceptions occur due to existence of larger key/value sets, where the limit comes from that the bottleneck of the leader, whose network is limited to only 7mb/sec. Python supposedly doesn't scale well across threads for CPU-bound tasks, however, its effect is mostly unnoticed during our tests, we suspect that 25/31 servers aren't high enough for the issue to surface.

However, there is still room for improvement: For our testing, we could increase the network speed for each AWS instance and check the real performance bottleneck of our implementation, when its not limited by network speed. For our implementation of RAFT, asynchronization needs to be implemented for our servers; GO or other languages are desired to increase our server side performance.

REFERENCES

[1] *Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm (extended version)." 2018-12-18].*, https://raft. github. io/raft. pdf

[2] *RAFT dissertation*, https://github.com/ongardie/dissertation

[3] *gRPC*, https://grpc.io/