CS754 P3: Raft optimization

Lingyi Zhang
l367zhan@uwaterloo.ca

Yuxin Fan
fyuxin@uwaterloo.ca

ABSTRACT

We look at the implementation of Logcabgin, one of the open source implementation of Raft consensus algorithm and the first implementation of Raft algorithm, and understand how it works. After looking into operation path of both read and write request in the LogCabin implementation, we are able to identify one potiental reason that slows down the read operation. We implement an change to the read request path that improve the total roundtime of each read request from the client side by 50% on average.

INTRODUCTION

Open source implementations of Raft are notoriously slow. This project will look into characterizing Raft operation path and identify which parts make Raft slow. Then we experiment one way to improve the effieiency of the Raft implementation.

Raft

Raft [2] is a consensus algorithm for managing a replicated log. Consensus algorithms play a key role in building reliable large-scale software systems since thet can allow a collection of machines to work as a coherent group that can survive the failures of some of its members.

Raft has the key elements of consensus, including leader election, log replication, and safety, and it also enforces a stronger degree of coherency to reduce the number of states that must be considered. It is claimed easier to understand compare to many other consensus algorithms. Besides, it also introduces a stronger leader and includes a new mechanism for cluster membership changes.

Raft, like many other consesus algorithms, maintain a replicated state machines on a collection of servers that compute identical copies of the same state and can continue operating even if some of the servers are down. Each server stores a log containing a series of commands, which its state machine executes in order. Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

Raft implements consensus by first electing a distinguished leader, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. A leader can fail or become disconnected from the other servers, in which case a new leader is elected. Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- Leader election: a new leader must be chosen when an existing leader fails.
- Log replication: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own.
- Safety: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index.

LogCabin

LogCabin is a distributed system that provides a small amount of highly replicated, consistent storage. LogCabin use Raft consensus algorithm internally and is the very first implementation of Raft. The implemention of Logcabin have the following major components: Tree, RPC, Storage, Event, Client and most important the Server. The code layout of Logcabin can be found in **Figure 1**. [1]

Tree is the core data structure for clients. All the data are in the format of key value pairs and stored in a **Tree** structure. The data is managed as a directory. LogCabin takes a snapshot of the **Tree** structure for each epoch. and data that transmit between client and servers send the snapshot around.

The raft consensus algorithm is implemented in **Server**. Raft divides time into terms and elects a leader at the begining of each term. This election mechanism guarantees that the emerging leader has all commited log entry. Once a candidate receives a vote from majority, it

becomes the leader and replicate its own log entry to followers. The leader is the only server that client talk with.

RPC implements a Low-level framing protocol to transmit data. It provides the application-level connection initiation timeout and heartbeats for leader election and maintain client sessions. Each service handles a related set of RPCs from clients. Thread dispatch for most services.

Storage is an in-memory and on-disk log. It persists a log on the filesystem efficiently. The log entries on disk are stored in a series of files called segments, and each segment is about 8MB in size. Thus, most small appends do not need to update filesystem metadata and can proceed with a single consecutive disk write.

Event contains an event loop based on Linux's epoll interface. It keeps track of interesting events such as timers and socket activity, and arranges for callbacks to be invoked when the events happen.

Client is responsible for resolving a DNS name, initiating a TCP connection, and waiting for an RPC response. The Client use LeaderRPC to send request to the leader of the LogCabin cluster. It automatically finds and connects to the leader and transparently rolls over to a new leader when necessary.

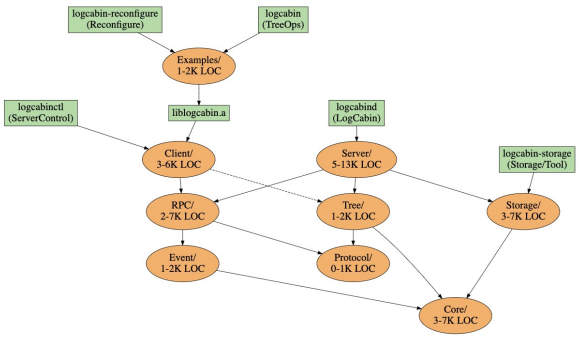

Figure 1. Code layout of Logcanbin

OUR FINDINGS

The goal of this project is to improve the implementation of raft consensus algorithm of LogCabin. The first thing we do is to understand the workflow of each client read and write request. We also mesaure the time it takes for each operations along the path.

Workflow

LogCabin was developed with clusters of servers in mind and where a single leader handles all client requests. The client library is structured so that most RPCs go through a LeaderRPC class to invoke operations on the cluster leader. The leader-based cluster RPCs were so pervasive that there was no way to ask a particular server to do something.

1. read request

A workflow of a client read request can be found in **Figure 2**. When a client sends a read request, the **LeaderRPC** on the client side will send a RPC call to the server. On the top of server side, **Loop::runForever** keeps listening request from client, each request is regarded as an event. The server handles one event at a time.

Then the **ClientService** processes client RPC call by **ClientService::handleRPC**. It will execute **ClientService::stateMachineQuery** which calls **RaftConsensus::getLastCommitIndex** to find the log entry. It returns the most recent entry ID that has been externalized by the replicated log. This is used to provide non-stale reads to the state machine. It would check whether the entry's term at commitIndex matches its current Term. So the server could know if it has up-to-date commitIndex which means it is the leader at the current term.

If it is the leader, it calls **StateMachine::wait** to make sure the state machine has applied at least the given entry. After that, the server calls **StateMachine::query** to execute read-only queries on the state machine and sends the reply to client. The workflow of the read can be concluded as follows:

(a) Client sends read request to one server in cluster(not the leader).

(b) The server receives request. It calls state machine to find who is the leader and responds client with leader info.

(c) Client receives response and resend request to the leader server.

(d) The leader receives request and it call state machine to check if it still the leader at that time. If yes, the leader queries state machine and replies the client; if no, the leader returns up-to-date leader info.

(e) Client receives the response.

2. write request

The workflow for write request is shown in **Figure 3**. When a client send a write request, the **LeaderRPC** on the client side will send a RPC call to the server. There are two commands before and after the first writing.

The first command is to open session. The servers' state machines keep track of client sessions. The client sessions are used to avoid applying an operation more than once if a client retries it. For example, if the client didn't receive a reply. Logcabin uses **ClientImpl::ExactlyOnceRPCHelper** to assign sequence numbers to RPCs, which servers then use to prevent duplicate processing of duplicate requests.

Once the session is created, the client sends the second RPC call which is the actual write operation. Once the server receives the request, it process request in **ClientService::stateMachineCommand** which handles RPC call from client. It firstly submit an operation

to the replicated log. If the cluster accepts this operation, then it will be added to the log and the state machine will eventually apply it. If server is the leader, then it call **StateMachine::waitForResponse** to get a response for a read-write command on the state machine. If the return value is true, the response will be filled in here. Otherwise, this will be unmodified. After that the server reply client with command result.

In the end, if there is no request needs to be sent, the client sends an RPC call to close the session. The workflow of write request can be concluded as follows:

(a) Client sends write request to one server in cluster (not the leader).

(b) The request is wrapped by LeaderRPC and divied into three RPC call with different commands.

(c) The first call contains open seesion command which creates session between server and client.

(d) The second call is to execute tree command which is the actual write operation. It applies this operation into its own log entries and replicate it into state machine. Once the operation is written in state machine, the operation is executed in all servers.

(e) The last call is sent from client after the last request. If the client has no request it will close the session then.

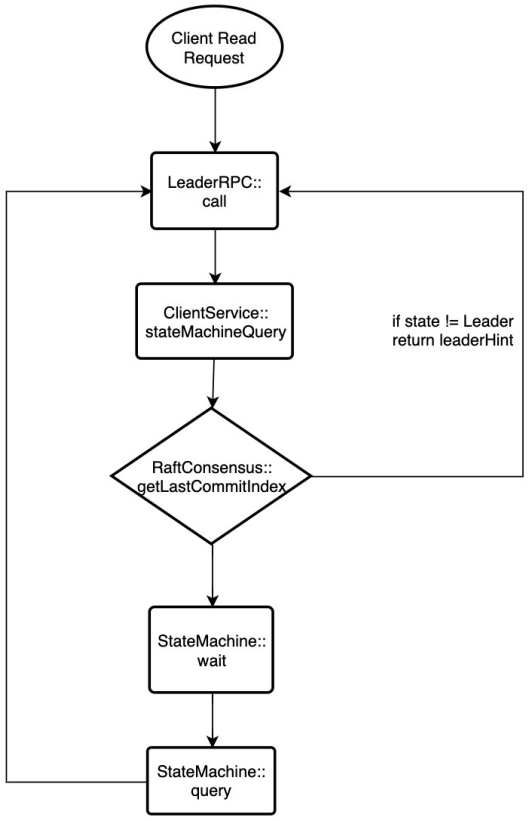

Figure 2. Processing read request in Logcabin

Benchmark

In order to figure out the most time-consuming part when processing the read request, we perform a benchmark measurement on our three-server cluster.

We randomly generate 1000 objects with different keys and values. Firstly, we write them one by one, after each writing we read the same key and calculate the average latency of each stages in read request as **Figure 4** shows. According to it, we find that two state machine query occupies 62% of total time used. Then we run the benchmark again and calculate average latency in operations of state machine query to figure out which part contributes to the high latency. The result is shown in **Figure 5**.

From the above analysis, we can see that the majority of the time needed to serve a client read request(or state machine query) is spent on the function **RaftConsensus::getLatestCommittedIndex**. This operation occupies almost 95% of the state machine query. This function calls **RaftConsensus::upToDateLeader** to execute non-stale read operations to client. It gives up after ELECTION_TIMEOUT, since leader will call **RaftConsensus::stepdown** and return to the follower state after the timeout. That is, the server have to talk to every nodes in the cluster and collect reponses.

OUR IMPROVEMENT

We focus on the read operation of the LogCabin implementation, and experiment our improvement. After apply our change, the roundtime of read request send from client side reduce to 50% of the original results.

High Level Design

From the above section, we identify the bottleneck for read request is to get the last commited index that used to read from the state machine. In LogCabin implementation, before the leader is able to read the index from its log, it sends out heartbeats to all followers to make sure it is still the leader. This process is necessary since before the read, it needs to make sure it is still the current leader before giving the index. Since this process need to send RPC request to all followers and wait for response, it dramatically slows down the roundtime of the read request.

However, since leader is the only server that have the authority to write the log, the commit index in its memory is equal to the last commit index across the cluster. The main idea for our improvement is to let the leader read the index from its local memory instead of talking to all other servers.

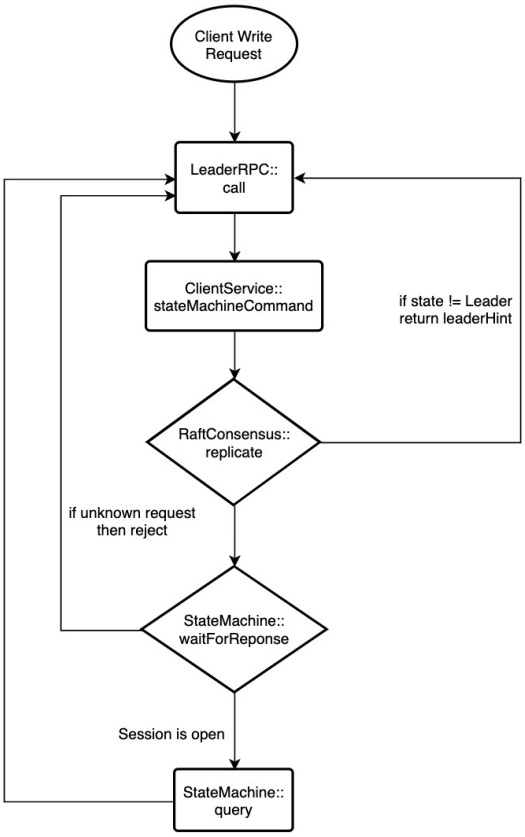

Figure 3. Processing write request in Logcabin

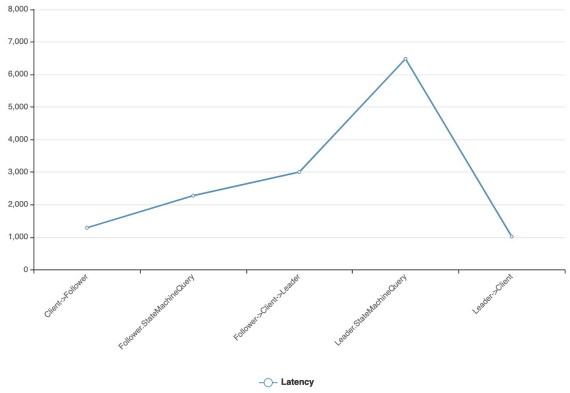

Figure 4. Average latency of read request in different stage

In order to guarantee we receive the right index after the change, in our change, we also need to make sure that the leader we talk to for the read request is still the current leader. This can be done by assigning lease to all the followers with heartbeats and stop the followers becoming leader during the lease period. Besides that, we turn the leader to follower when it is not able to grant it lease to majority of the servers in the cluster. On the other hand, there's a chance a new leader is elected.

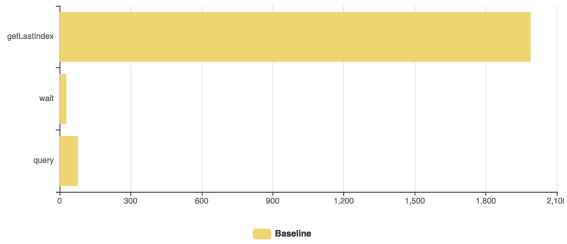

Figure 5. Average latency of state machine query in different state

In order to improve the efficiency of read request and maintain the consistency, we do the following changes:

- Leader reads the latest index from local memory
- Followers reject all vote requests within the period of two heartbeat interval
- Leader turns into follower when two heartbeat does not reach to majorities of the servers

This change does not affect raft's solution to the three subproblems that the raft raise for conensus problem.

- Leader election: We still relay on the original raft consensus algorithm to do the leader election. When the current leader stepdown, a new leader will be elected through the leader election.
- Log replication: A read request do not change the data store in the tree, so we do not need to log any changes.
- Safety: Read request does not affect the log replication, so our change also would not affect the safety.

After applying this change, we will expect to see a big improvement in read request roundtime on the cost of slowing down the raft server communication a bit. Since we simplify the steps that the LogCabain need to serve the client read request, the client read request will behave better. However, we achieve that on the cost of slow down the process that servers send and handle hearbeats. This is will not cause a huge performance issue since LogCabin exchange regular heartbeats among the servers on the other threads which does not block the read requests anymore.

Implementation

After digging the LogCabin implementation, we notice that each server that running **RaftConsensus** record its state and the leader record the latest commit index in the field **commitIndex**. We create a new function **RaftConsensus::getCurrentState** that return the current state of the server and the commited index if the server is the leader. We change the function in **ClientService::stateMachineQuery** from calling **RaftConsensus::getLastestCommitedIndex** to **RaftConsensus::getCurrentState**, which read the index from the memory.

Each server also maintain a field **RaftConsensus::withholdVotesUntil** to decide whether it will reply to the vote. For the follower, this value will be increment by two heartbeat intervals each time it receive a heartbeat. For the leader, this value is set to **timepoint::max**. During each **RaftConsensus::HandleRequestVote**, the server will reject the vote if the clock is less than the **RaftConsensus::withholdVotesUntil**. In this case, we can ensure that any server who receives heartbeat will not response to vote within two heartbeat interval. Thus, the leader knows that there will be no new leaders if the majority of the servers in the cluster are not behind for more than two heartbeats.

Finally, we need to make sure that the leader stepdown when there might be potential new leaders. This is done by checking the lastest heartbeat the majority servers in the cluster receives. In LogCabin, heartbeat is sent as an **appendEntry** request from the leader and each time the leader send out **appendEntry** request it also increment the term number. The follower who receives the **appendEntry** request from the leader will execute **RaftConsensus::handleAppendEntries** process. For each **RaftConsensus::handleAppendEntries**, the term of the follower will increment to the term number sent by the leader. In this case, we can use term number to determinate the most recent heartbeat that the follower has received.

Everytime, the leader send out a heartbeat, it checks the majority of the term number among the cluster. If the majority of the servers in the cluster has the term number larger than the leader's current term number minus 2, then the leader knows that the majority of the servers in the cluster are not behind for more than two heartbeats and no potential leader in the cluster. Therefore, it is safe for the leader to continue be the leader and excute the request. Otherwise, the leader will call **RaftConsensus::stepdown** and turn into a follower.

Test

In order to ensure the consistency of the raft consense model after our modification, we need to make sure we have enough test coverage. After having a throughout check on the existing test cases in the LogCabin implementation, We do some modifications on existing the test cases to reflect our change of the implementation. In addition, we also implement some new test cases to ensure that no leader is elected during the lease period and the test can be found in **RaftConsensus:**.

Performance

Our optimization performs well in terms of processing read-only queries. We build benchmark with 1000 objects and calculate latency on average, maximum, minimum, 25th, and 75th percentiles for read request and comparing them with original implementation. The result is showing on **Figure 8**. The roundtime from

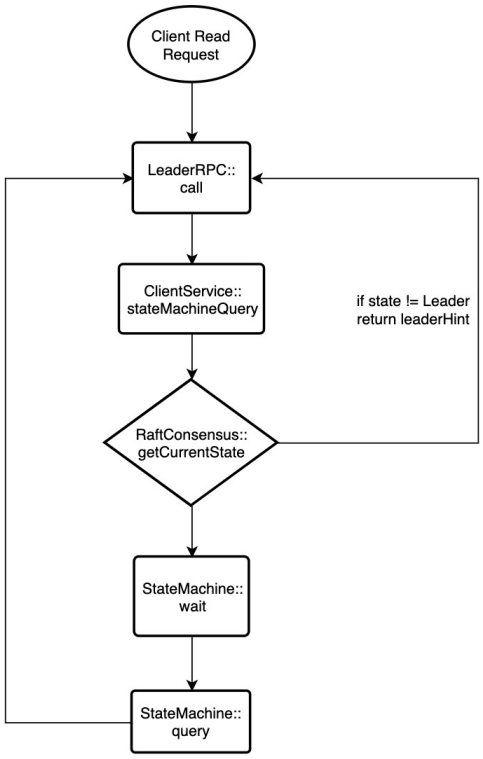

Figure 6. Processing read request on improved Logcabin

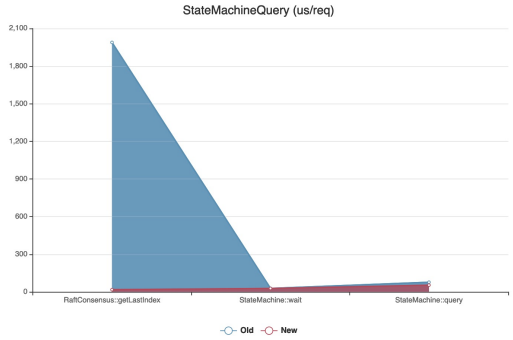

Figure 7. Latency in different stages of StateMachineQuery

client side of each read request reduced 50% on average. For original implementation, it spends $4656\ us$ on read request. After applied our improvement, it takes $2098\ us$ to finish the read request on average. As the result shown in **Figure 9**, we reduce the running time of **ClientService::stateMachineQuery** from $2190\ us$ to $469\ us$ on average. We could determine that our optimization indeed improved the perfomance of Logcabin.

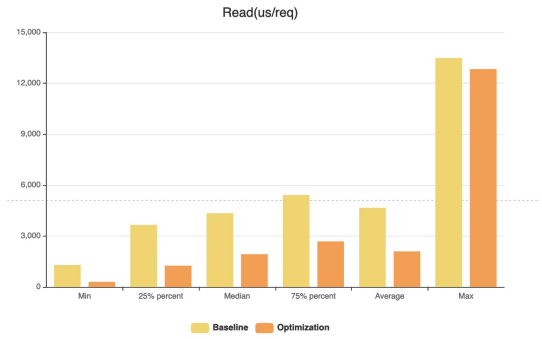

Figure 8. Analysis of read request

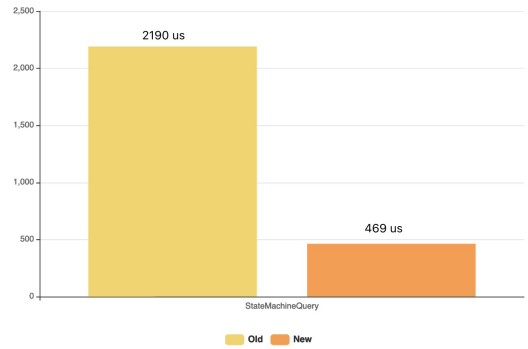

Figure 9. Comparision of time-consumed in StateMachineQuery between original and improved Logcabin

APPENDIX

- **Environment**

  This project is running on Docker. We setup three dockers and they share the same ip address but use different port. The Dockerfile is attached in *logcabin.zip*.

- **Deployment**

  We run three-servers cluster on our own laptop.

- **Debugging**

  To figure out how Logcabin works, we use **GDB** for command line debugging. It allows us to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

- **Testing**

  On the basis of **gtest**, we add **RaftConsensus::rejectVotesAfterHeartbeat** to test our changes.

REFERENCES

1. Diego Ongaro. Logcabin https://github.com/logcabin/logcabin.

2. Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.