Assignment 2: Raft Implementation

Distributed Data Processing Systems

Group 11: Wei Chen, Yuxuan Zhu

1 Introduction

In this assignment, we will be implementing Raft, a replicated state machine protocol, in Java. A replicated service is a type of service that achieves fault tolerance by storing complete copies of its state on multiple replica servers. This way, even if some of the servers experience failures, the service can continue to operate by using the other replicas. The challenge with replication is that failures can cause the replicas to hold differing copies of the data, which can lead to inconsistencies in the service state.

Raft addresses this challenge by organizing client requests into a sequence called the log, and ensuring that all replica servers see the same log. Each replica executes client requests in the order they appear in the log, and applies them to its local copy of the service state. Since all the live replicas see the same log, they all execute the same requests in the same order, which ensures that they all have the same service state. If a server fails but later recovers, Raft takes care of bringing its log up to date so that it can continue to operate as part of the replicated service.

In this lab, we will implement Raft as a Java object type with associated methods. The goal is to use Raft as a module in a larger service to achieve fault tolerance through replication. We will use RPC to enable communication between the Raft instances, which will allow them to maintain replicated logs. Our Raft interface will support an indefinite sequence of numbered commands, also known as log entries. These entries will be numbered with index numbers, and the log entry with a given index will eventually be committed.

2 Raft Background

Distributed storage systems are often fault-tolerant. They improve system availability by maintaining multiple copies. To achieve this goal, it is necessary to solve the core problem of distributed storage systems: maintaining consistency across multiple copies. Consensus is the basis for building fault-tolerant distributed systems. In a consistent cluster, all nodes have the same result for a value stored in it at the same time, i.e., they are consistent about their shared storage. The cluster is auto-recovery in nature, so that when a few nodes fail it does not affect the normal operation of the cluster, and when most of the nodes in the cluster fail, the cluster stops serving (without returning an error result). Consistency protocols ensure that the system can still provide services to the public even if some (or rather, a small) number of replicas are down. Consistency protocols are usually based on replicated state machines, i.e., all nodes start from the same state, go through the same sequence of operations (logs), and finally arrive at the same state.

The most typical consistency protocols include Paxos, Raft, etc. Both Paxos and Raft are essentially single-owner consistency algorithms. But Paxos algorithm is more theoretical compared to Raft; the principle understanding is more abstract and only provides a set of theoretical prototypes, which makes it very difficult to understand and implement. In contrast, Raft is clearly described, and the authors list the implementation steps of the prototype algorithm in a complete paper, which greatly facilitates the implementation of the algorithm by engineers in the industry and thus has been more widely used in recent years.

Similar to other consensus protocols, there is only one Leader in the Raft cluster, all other nodes are Follower. the Follower are passive: they do not send any requests, but simply respond to requests from the Leader or Candidate. the Leader is responsible for handling all client requests. To simplify the logic and implementation, Raft decomposes the consistency problem into three separate subproblems.

(1) **Leader election**: When the leader is down or the cluster is created, a new leader needs to be elected
(2) **Log replication**: the leader receives requests from clients and replicates them in the form of logs to other nodes in the cluster, and forces the logs of other nodes to be consistent with itself
(3) **Safety**: If any node has applied a definite log entry to its state machine, then no other service node can apply an unused command at the same log index location.

2.1 Leader Selection

When a cluster with Raft protocol is first started, all nodes are in the Follower state. Since there is no Leader, the Follower cannot keep a heartbeat with the Leader and the Follower waits for the heartbeat to time out (each Follower has a different heartbeat timeout). Therefore, Followers will think that the Leader is dead. The first Follower to time out then moves to Candidate status, and Candidate then asks the other nodes in the cluster to vote to upgrade itself to Leader, and if Candidate receives votes from more than half of the nodes, it will be selected as Leader.

In the implementation, when a cluster using the Raft protocol is first started (or when the Leader is down and restarted), all nodes are in the Follower state, with an initial Term (the unique identifier for a given election) of 0. The election timer is started at the same time, and the election timer for each node is inconsistent and between 100 and 500ms (to avoid simultaneous election initiation).

If the node does not receive a heartbeat and a poll request within one election timer cycle, the state is changed to Candidate, the Term is incremented, and a poll request is sent to all nodes in the cluster and the election timer is reset.

The election timer timeout for each node is within 100 to 500 ms and is inconsistent. In other words, the first node to convert to Candidate and initiate a polling request will have the first-mover

advantage of becoming the Leader.

When a Follower node receives a VoteRequest RPC, it decides whether to accept the request based on the following conditions.

- If the requesting node's Term is greater than its own and it has not yet voted for another node, it accepts the request and votes for the Candidate node.
- If the requesting node's Term is smaller than its own Term and it has not yet voted for itself, it rejects the request and votes for itself.

After one round of election, normally, one Candidate node will receive more than half (N/2+1) of the votes from other nodes, then it will win and be promoted to Leader node, then it will send heartbeats to other nodes at regular intervals, and the other nodes will be transformed into Follower nodes and keep in sync with the Leader, so that the round is over. If the Candidate node does not receive more than half of the votes in a round, then the next round of elections will be held.

2.2 Log Replication

One important feature of Raft is its strong leader. Only the Leader node in a Raft cluster can handle client requests (if a client request is sent to the Follower node, the Follower will redirect the request to the Leader), and each client request contains an instruction that is copied to the state machine for execution. The Leader appends this instruction to the log as a new log entry and then sends the additional entry in parallel to the Followers for them to copy the log entry. When the log entry has been safely copied by the Followers, the Leader applies the log entry to its state machine and returns the result of the execution to the client. If the Follower crashes or runs slowly, or if the network drops packets, the Leader will keep trying to append the log entry repeatedly (despite having replied to the client) until all Follower have finally stored all log entries, ensuring strong consistency.

In the first stage of the implementation, the Leader receives a request from the client to, for example, store a piece of data, and when it receives the request, it writes it to the local log as a log entry. At this point the entry is uncommitted and the Leader does not update the local data, so it is not readable.

In the second stage: the Leader sends the entry to the other Follower, the Leader maintains a heartbeat with the Followers and sends the AppendEntries to the other Follower nodes in parallel with the Heartbeat Leader and asks them to copy the log entry, a process we call Replication.

In the third stage, the Leader waits for a response from the Followers, who receive the replication request from the Leader and have two possible responses.

- Write to the local log and return Success
- Consistency check failed, write rejected, false returned.

The status of the Entry is also uncommitted at this point. Once the above steps have been completed, Followers will send a response to the Leader - success - and when the Leader has received the majority of responses from Followers, it will mark the Entry written in the first stage as committed and apply this log entry to its state machine.

In the fourth stage, the Leader responds 'True' to the client which indicates the writing operation was successful.

During the last step, the Leader notifies Followers that the Entry has been committed. After the Leader responds to the client, it will notify Followers with the next heartbeat, and Followers will also mark the Entry as committed when they receive the notification. At this point, more than half of the nodes in the Raft cluster have reached a consistent state, ensuring strong consistency.

2.3 Safety

The follower or leader may crash at any time, so Raft must continue to support log replication in the event of downtime and ensure consistent log order for each replica, with three possibilities in total

(1) usually because of network delays follower does not leader response, then the leader will constantly reissue Append Entries RPC
(2) the follwer crashes and then recovers. the consistency check of the Raft append entry takes effect, ensuring that the follower recovers the missing logs in order after the crash.
(3) If the leader crashes, the crashed leader may have copied the logs to some follower, but has not yet committed them. It is possible that the new leader selected at this point does not have these logs, so that some of the logs in the follower are not the same as the logs in the new leader. In this case, the leader resolves the inconsistency by forcing the follower to copy his logs.

In the implementation, the prevLogIndx and prevLogterm parameters in AppendEntries are used to perform consistency checks. Specifically, the leader puts the Index and Term of the previous log entry in each AppendEntries RPC sent to the follower. if the follower does not find the previous log from the leader in its own logs (based on the Index and Term), then it rejects the log. The leader then receives a rejection from the follower and sends the previous log entry, thus gradually moving forward in a recursive way to locate the first missing log from the follower.

3 Implementation

In our implementation, we mainly focus on the *leader election* and *log replication* of the Raft protocol (which are the two most important parts), due to the time limitation and the protocol's complexity. The protocol is deployed on DAS-5. We implemented the two prominent RPCs in the original article, as illustrated in Fig.11 and Fig.2, to enable election and log replication. We use Protobuf [2] as a serialisation framework to facilitate data storage and communication. As Raft is to be tested on a distributed platform, it is also vital to build the communication protocol. In this assignment, gRPC [1] is used for communication between nodes.

3.1 Implementation Rules

To realized the basic features of Raft, the following rules should be obeyed.

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 1: RequestVote RPC for Raft

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

Figure 2: AppendEntries RPC for Raft

- There can only be one Leader per election
- Leader's previous log entries can not be changed. Only add new enties are allowed.
- Same Index and Term for the same Entry
- If a log is committed by a Leader, the subsequent Leader must have that log entry
- If a node has applied an entry to its state machine for a given index, no other server can apply a different entry to that index

The different states of a node will determine its behavior, and the state change of a node will be performed through RPC.

Therefore, we can summarise the behaviour and communication of the Follower, Candidate and Leader according to the original paper, and use this as a rule for implementation.

3.1.1 Follower

- Listening for log entry requests and messages from the Leader
- Forwards requests for log entries to the Leader
- Append log entries when a commit message is received from the Leader
- If the Follower does not receive a message from the Leader,it will turn itself into a Candidate and send a vote request to all nodes in the cluster.
- Follower's heartbeat will indicate that they are alive and tell the Leader their highest committed index.

3.1.2 Follower RPC

- Respond to Leader heartbeat (appendEntries)
- If another node triggers an election, then the follower may receive an RPC for a vote request that will contain the highest commit index of the sender, and the follower responds to the node.

3.1.3 Candidate

- Send the vote request to each node in the cluster and wait for a response
- If more than half of the votes are received (N/2+1), count the votes and declare itself as Leader

3.1.4 Candidate RPC

- VoteRequest

3.1.5 Leader

- send heartbeat (appendentries) to all cluster members, the heartbeat msg needs to be repeated during idle periods as well to prevent triggering unnecessary elections
- If the leader receives a log request from a client, it appends the entry to its local log and responds to the requesting node after the entry has been applied to the state machine

3.1.6 Leader RPC

- Leader heartbeat
- Submit Entry
- AppendEntries

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 3: RequestVote RPC for Raft

3.2 Communication

3.2.1 ProtoBuf

For inter-process communication, we always first consider data transfer between processes. Data must be serialized before transfer. Google's ProtoBuf is introduced in our project for data serialization. Specifically, data that is to be accessed and manipulated by both the client and the server will be serialized. For example, **RequestVoteRPC** is issued by a raft server to send to another. the former raft server is now a client that calls services and that carries its arguments, while the latter is now treated as a server that provides services, with respect to RPC mechanism. In this case, we serialize a class (or message used in Protobuf) called, say **RequestVote** for transfer.

3.2.2 gRPC

gRPC is also developed by Google and is perfectly compatible with **Protobuf**. It is a remote procedure calling framework that transfers serialized data between processes. In practice, after defining and serializing data (encoded as *.proto* file) used for transfer, we design the consensus service and its corresponding calling method for the raft server since a raft server is both a server and a client during its lifecycle.

4 Experiments

In this experiment, we use das-5 as the platform to test Raft. We set timeout as 5-15s, the thread pool size is 20, and heartbeat is 500ms.

4.1 Leader Election

In this experiment, we test the leader election performance for 3,5, and 7 servers clusters, shown in Table 1. We mainly focus on whether the leader can be selected in certain time.

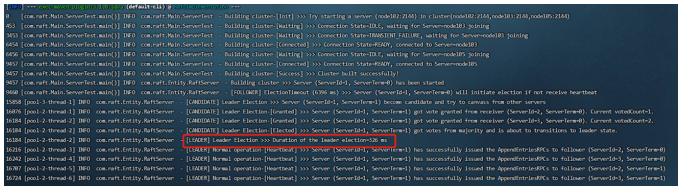

Figure 4: Three nodes illustrator

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 5: RequestVote RPC for Raft

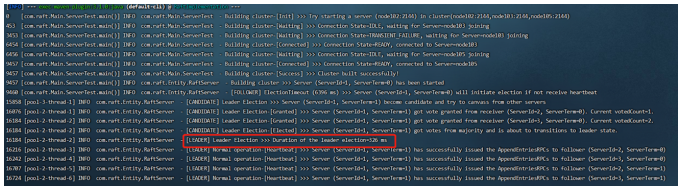

Figure 6: Three nodes illustrator

3 servers clusters	5 servers clusters	7 servers clusters
413	518	671
477	573	642
482	561	651
412	482	590
449	599	610
442	543	623
478	580	698
419	519	679
462	440	612
409	537	648
412	581	690

Table 1: Election time of different clusters

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 7: RequestVote RPC for Raft

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 9: RequestVote RPC for Raft

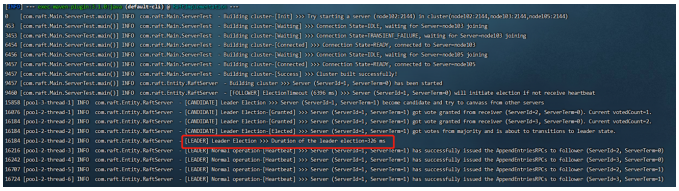

Figure 8: Three nodes illustrator

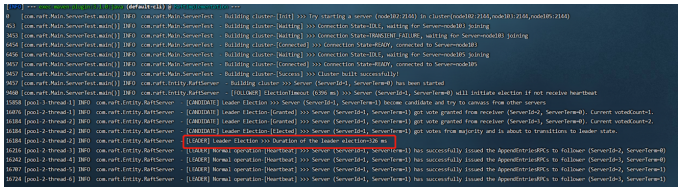

Figure 10: Three nodes illustrator

Through running the raft.sh, we can get the election time. For each experiment, we perform it for 10 times, in avoiding the influence of system wave.

Through the experiments, we can find that our raft implementation is able to communicate through each node. With the increasement of cluster size, the election time will grow a little bit but still in a normal range. The results obey the rules we mention in previous chapter. In the same period, only one server can be elected.

4.2 AppendEntries

We ran the log part on a 3-nodes raft cluster. After starting node102, node103, node105, we send an entry from node106 (client)

One node received the request, but it's not the leader, so it redirect the request to leader.

Leader then send AppendEntries RPC to each follower. After receiving the agreement form most of the followers, leader will commit the entry. In this experiment we commit the request for twice, so the index is 2 and the term is 1.

5 Conclusion

In our project, we implemented (not fully) the core problem of Raft algorithm: Leader election, log replication, and safety. However, we could concluded that safety is the most important point we must take into account in terms of development since there were various exceptions occurring when overlooking the implementation of safety check.

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Figure 11: RequestVote RPC for Raft

References

[1] [n. d.]. GRPC. https://github.com/grpc
[2] [n. d.]. Protocol buffers nbsp;|nbsp; google developers. https://developers.google.com/protocol-buffers