RAFT (Log Replication)

DNP lab 7 @ Innopolis University, Fall 2022

Task

In this assignment you have to implement another part of RAFT protocol.
This part is a Log Replication
To do so, **you still must use gRPC**.

You can do this task in a team of 2 people.
By default, you are in the same teams as on the previous lab. If you want to change your team, do it here (This is a new copy of the previous document).

`IMPORTANT`: Your output messages **must be strictly** the same as given in the task. Their structure, not order.

Submission

A single `.zip` archive containing:

1. `server.py`
2. `client.py`
3. `raft.proto`

All 3 files are in the same directory.

Log

Log is a list of entries.

Each entry consist of:

1. index
2. term number
3. command

Server

Every server have:

- commitIndex - index of the last log entry on the server. 0 at the start.
- lastApplied - index of the last applied log entry. 0 at the start.

If commitIndex > lastApplied: increment lastApplied and apply log[lastApplied] to state machine.

Only leader:

Reinitialized after election

For every node address leader has two fields:

- nextIndex - list of indexes of the next log entry to send to each server. Initialized to leader last log index + 1.
- matchIndex - list of indexes of highest log entry known to be replicated on each server. Initilized to 0.

RequestVote

Now this function checks that only the node with the most complete logs can become the leader.

`RequestVote(term, candidateId, lastLogIndex, lastLogTerm)`

`term` - candidate's term.
`candidateId` - id of a candidate.
`lastLogIndex` - index of candidate's last log entry.
`lastLogTerm` - term of candidate's last log entry.

This function should return two values:

1. term number of the server
2. result of voting (True/False)

The result:

1. If `term` < term number on this server, then reply False.
2. If this server already voted on this term, then reply False.
3. If `lastLogIndex` < last log index of the server, then reply False.
4. If there is lastLogIndex entry on this server, and its term is not equal to `lastLogTerm`, then reply False.
5. Reply True

Note: if the `term` > term number on this server, you still should update it

---

AppendEntries

`AppendEntries(term, leaderId, prevLogIndex, prevLogTerm, entries, leaderCommit)`

`term` - current term number from the leader.
`leaderId` - leader's id. So that the follower knows who his leader is.
`prevLogIndex` - index of previous log entry (immediately preceding new ones).
`prevLogTerm` - term of prevLogIndex entry.
`entries` - a list of new entries to store. Empty for heartbeat.
`leaderCommit` - **commitIndex** of the leader.

This function should **return two values**:

1. term number of the server
2. success (True/False)

Algorithm:

1. If term number on this server > `term`, return False
2. If log does not contain an entry at prevLogIndex, return False
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it.
4. Append any new entries not already in the log.
5. If `leader commit` > **commitIndex**, set commitIndex = min(leaderCommit, index of last new entry)

If, as a result of calling this function, the Leader receives a term number greater than its own term number, that Leader must update his term number and become a Follower.

For a leader

- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex.
  - If successful: update nextIndex and matchIndex for the follower.
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry.
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set **commitIndex** = N

SetVal

`SetVal(key, value)`

`key` - a key to store the value. String.
`value` - value itself. String.

Returns:

1. `success` - True/False

This function is called by the client.

If this function is called on the leader, add a `key=value` entry to the log. When the entry is successfully replicated, apply it, store the key value and respond with True to the client. If something went wrong, reply False.

If this function is called on the follower, redirect this call to the leader.
If this function is called on the candidate, reply False. *Optionally you can block until leader is elected and then redirect this call to it, as it was initially designed in RAFT.*

GetVal

Don't append a new log entry on this function call!

This function is called by the client.

`GetVal(key)`

`key` - a string key to get value.

Returns:

1. `success` - result of operation. True/False
2. `value` - a string value.

Simply returns the value, if it is commited to the distributed system. `success` is True.
If there is no such key in the storage, reply with `success` is False.

Client

Add 2 new commands:

`key` and `value` are words (strings without whitespaces)

1. setval <key> <value> - call SetVal(key, value) on the connected server, print the result.
2. getval <key> - call GetVal(key) on the connected server, print the result and the value.

Example

`Config.conf` file:

```
0 127.0.0.1 50000
1 127.0.0.1 50001
2 127.0.0.1 50002
```

After starting three nodes one of them would be elected to be a leader. Example scenario from previous assignment:

```
>python server.py 0
The server starts at 127.0.0.1:50000
I am a follower. Term: 0
The leader is dead
I am a candidate. Term: 1
Voted for node 0                  // Voted for itself
Votes received
I am a leader. Term: 1
Command from client: getleader
0 127.0.0.1:50000                 // This result is also sent to the client
```

Server B:

```
>python server.py 1
The server starts at 127.0.0.1:50001
I am a follower. Term: 0
```

```
Voted for node 0
I am a follower. Term: 1
```

Server C:

```
>python server.py 2
The server starts at 127.0.0.1:50002
I am a follower. Term: 0
Voted for node 0
I am a follower. Term: 1
```

Leader must commit entries (and make changes visible via GetVal) only when it gets confirmation from 50% + 1 nodes. Let's terminate server C, connect to server B and ask to save key. It should work, since leader (server A) still have majority of nodes alive (2 out of 3):

At this point server C must be terminated (Ctrl + C). Then client would give us following output:

```
>python client.py
The client starts
> connect 127.0.0.1 50001
> getleader
0 127.0.0.1:50000
> getval key1
None
> setval key1 100
> getval key1                             // After 2 seconds
100
```

127.0.0.1:50001 node is follower and redirects setval requests to leader, and answers getval itself.

Now, if we terminate server B, leader server A wouldn't have 50%+1 confirmations to commit changes:

```
>python client.py
The client starts
> connect 127.0.0.1 50000
> getleader
0 127.0.0.1:50000
> getval key1
100
> setval key1 200                         // This change is not commited
> getval key1                             // After 2 seconds
100
> setval key2 300
> getval key2                             // After 2 seconds
None
```

Now if we start server B back, after some time leader would have enough confirmations and changes would be commited.

```
>python client.py
The client starts
> connect 127.0.0.1 50000
> getval key1
200
> getval key2
300
```