RAFT Protocol

Lab08@DnP

1. You should submit at least two files: `server.py` and `client.py`.
2. All archives in a `.zip` format.

Term

The life cycle of the system is divided into terms. Each term starts with the election. Term number is starting at 0 and increases each term.

Client

Has no command line arguments. Can handle `KeyboardInterrupt`

Can handle the following commands from the user:

1. `connect <address> <port>` - sets the server address and port to the specified ones. It does not connect anywhere, but `set` and `get` commands will be now sent to this address.
2. `set <string> <number>` - requests server to set the key `string` to the value `number`. Prints `True` or `False`.
3. `get <string>` - requests the value of the key `string` from the server. Prints this value, or `False`.
4. `quit` - exits the program.

If the server is unavailable, it should print `The server <address>:<port> is unavailable.`.

Also, it should print `The client starts` at the begging. And `The client ends` when it shuts down.

Example:

```
>python client.py
The client starts
>connect 127.0.0.1 50000
>set Hello 5
True
>set Hello 7
True
>get Hello
7                           // at this point we shutted down the server
>set World 10
The server 127.0.0.1:50000 is unavailable.
>connect 127.0.0.1 50001
>set World 10
True
>quit
The client ends
```

Configuration file

Contains information about the system in the format: `id address port`. The number of lines means the number of servers (nodes) in the system. Has the name `config.conf`. Located in the same folder with `server.py`.

Example:

Config file with five nodes:

```
0 127.0.0.1 50000
1 127.0.0.1 50001
2 127.0.0.1 50002
3 127.0.0.1 50003
4 127.0.0.1 50004
```

Server

Has one command line argument: `id`.

Startup

At the start, server reads the config file, finds the corresponding address and the port number, and binds to it. Also, it prints the address and port to which it is bound.

Example:

```
python server.py 1
Server is started at 127.0.0.1:50001
```

In addition, it remembers all the other servers, for further communication with them, and counting votes.

Also it handles its `term number`. Term number is initialized to 0 at the beginning, and increases by 1 in each election round.

States

The server has its own `heartbeat timer` with a randomly selected time from the range `[150, 300]` milliseconds.

`Follower` - the initial state of the server.

1. Every time Follower receives a message from the Leader (heartbeat or other), it resets the timer to a new random period.
2. If it receives a `RequestVote` message with a term number greater than current leader's term, it should vote for a given Candidate, if it has not already voted in this term.
3. If the timer expires, the Follower becomes a Candidate.

`Candidate` - a server which wants to become a Leader. When a Follower becomes a Candidate, it **increments its term number**, starts an election, votes for itself, and sets a new random timer. It requests votes from all other servers using RequestVote RPC message. Possible outcomes:

1. Candidate receives votes from majority of servers. It becomes a Leader and starts to send periodic AppendEntries RPC messages to everyone
2. Candidate receives AppendEntries RPC message
    - If term # in message >= than its own term #, the candidate accepts the sender as a Leader and becomes a Follower
    - Otherwise, it rejects the RPC and remains a candidate
3. The timer expires without majority of the votes. It generates a new random timer and becomes a Follower.

`Leader` - sends `AppendEntries` RPC message to all its Followers with the request from the Client. After every message, it resets its timer. If the timer is up, Leader sends an empty `AppendEntries` message (this is the heartbeat message), and, again, resets its timer.

Also, Followers should redirect requests from the Client to its Leader.

More about the Leader

The Leader is responsible for maintaining a distributed Log. When it receives a request from the Client, it:

1. Appends a new entry to its Log.
2. Sends `AppendEntries` message to all its Followers.
3. If the majority of Followers returned True, the Leader sends commit message to its followers, and return True to the Client.
4. If the majority of Followers returned False, the Leader returns False to the Client.
5. If some of the Followers are unavailable, the Leader will try to send them a message endlessly.

Functions

The server has the following RPC functions:

Request Vote

This function is called by the Candidate during the elections to collect votes.

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

`term` - candidate's term `candidateId` - id of a candidate `lastLogTerm` - term number of the last entry in the Log `lastLogEntry` - last Log entry

When the server receives this call, it should return `True` if:

1. The `lastLogTerm` equals to the term number of the last entry of log on this server.
2. The `lastLogEntry` equals to the last entry of log on this server.
3. `term` is greater than or equal to the term number on this server.
4. This server did not vote in this term.

If one of the above conditions is not met, it should return `False`.

Append Entries

This function is called by the Leader to add a new entry and to commit it.

`AppendEntries(term, leaderId, prevLogTerm, prevLogIndex, entry, commit)`

`term` - current term number from the leader `leaderId` - leader's id. So follower can redirect requests from client. `prevLogTerm` - the term number of the previous Log entry. `prevLogIndex` - index of the previous Log entry. New entry should be added after this index. `entry` - command itself. `commit` - boolean (True/False) value, which indecates is it a commit request or not.

When a Leader receives a request from a client, it calls this function on all of his followers with `commit=False`. Every follower should add check is it possible to execute command `entry` and return `True` or `False`. When the leader receives the majority of the approvals (`True` results) from the followers, it sends this message again with `commit=True` and runs the command itself. If the majority has not been reached, the leader considers the command not completed and sends `False` to the client When the Follower recieves this request with the `commit=True`, it runs the command itself.

It should return `True`, if:

1. `term` >= term number of this server.
2. `prevLogTerm` equals to the term number of the last entry of log on this server
3. There is a log entry with index `prevLogIndex`
4. `entry` command can be executed (e.g. there is a key `hello` for the command `get hello`)

If one of the above conditions is not met, it should return `False`.

If the `commit=True`, it should execute the `entry` command (e.g. set a key `hello` to the value `5` in its local hashtable if the command is `set hello 5`).

Execute

This function is called by the client.

`execute(command)`

`command` might be:

1. `set <string> <value>` - set the key `string` to the value `value`
2. `get <string>`- return the value of the key `string`

If this server is a Follower, it should redirect this request to the Leader. If this server is a Leader, it should start the command execution process using `AppendEntries` function.

This function should return `True` or `False` to the client. And an additional information if required (e.g. the result of `get` command).

Example:

`Config.conf` file:

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

Server A:

```
>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: set Hello 5
Check command: set Hello 5
True
Commit command: set Hello 5
Return True                         // From the execute function
Command from client: get Hello
Return (True, 5)
^C                                  // KeyboardInterrupt
The server ends
```

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
Command from client: set Hello 5
Redirect to the leader 0            // Current leader
Check command: set Hello 5
True
Commit command: set Hello 5
Return True                         // From the execute function
Command from client: get Hello
Redirect to the leader 0
Return (True, 5)
Voted for node 2
I am a follower. Term 2
```

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
Check command: set Hello 5
True
Commit command: set Hello 5
The leader is dead
I am a candidate. Term: 2
Voted for node 2                    // Voted for itself
Votes received
I am a leader. Term: 2
```

Client:

```
>python client.py
The client starts
>connect 127.0.0.1 50001
>set Hello 5
True
>get Hello
5
```