RAFT Protocol (Leader election)

Lab08@DnP v2

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.

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

Server has:

1. `term number` - 0 at the startup. Increases by 1 in each election round.
2. `timer` - initialized at the startup. Random value in the range `[150, 300]` ms.
3. Information about other servers. Their number, and their addresses.

Startup

Server has one command line argument: `id`.

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
```

States

`Follower` - the initial state of the server.

Every time Follower receives a message from the Leader, it resets the timer.
If the timer is up, Follower becomes a Candidate.
If it receives a `RequestVote` message, it should vote for a given Candidate, if it has not already voted in that term.

`Candidate` - trying to become a leader.

First, it **increments its term number** and resets its timer.
Then, it requests votes from all other nodes. Also, Candidate votes for itself.
If it has the majority of votes before the timer is up, the Candidate becomes a Leader.

If the timer is up, and the Candidate does not have the majority of votes, it generates a new timer (with the new time) and becomes a Follower.

If the Candidate receives the message (any message) with the **term number** greater than its own, it stops the election and becomes a Follower. Also, it should update its term number with received term in this case.

`Leader` - runs the system.

Every **50 milliseconds** sends an `AppendEntries` request to all other servers. This is the `heartbeat message`.

If the Leader receives a heartbeat message from another Leader with the **term number greater than its own**, it becomes a Follower (and also sets its term number to the new leader's one).

Functions

Server has the following RPC functions:

Request Vote

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

`RequestVote(term, candidateId)`

`term` - candidate's term number
`candidateId` - id of a candidate

This function should **return two values**:

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

The result:

1. If `term` is equal to the term number on this server AND this server did not vote in the given `term`, then result is `True`
2. If `term` is greater than the term number on this server, then:
    - update its term number with the `term`
    - check condtion 1.
3. Else, the result is `False`

If the server is in the Candidate or Leader state and receives this request and the result is True, it must become a Follower (its term number should be also updated during calculation of the result).

If, as a result of calling this function, the **Candidate** (the one who tried to collect votes) receives a term number greater than its own term number, that Candidate must update its term number and become a Follower.

Some clarifications:

1. Don't forget to **update the timer**, as the server updates it whenever any message is received.
2. If the server had term number 1, and received a RequestVote with term 2, it should raise its term number to 2, and vote for this candidate. If immediately after that, it receives a RequestVote with term 3, it must also raise its number to 3, and vote for the new candidate
3. If no heartbeat is received after this request and the timer expires, this server becomes a candidate.

Append Entries

In this lab, this function is only used to send heartbeat messages.
The name is left unchanged to understand that the same function is used to replicate the log.

```
AppendEntries(term, leaderId)
```

`term` - current term number from the leader
`leaderId` - leader's id. So that the follower knows who his leader is.

This function should **return two values**:

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

If `term` >= term number on this server, than `success=True`. Else, `success=False`.

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.

Some clarifications:

1. As mentioned above, Leader calls this function from all other servers every 50 ms.
2. When this function is called on the server, the server resets its timer.

GetLeader

This function is called by the client.

`GetLeader()`

Returns the current Leader id and address.

If there are elections now, the function should return information about the last node that this server voted for. If this server has not yet voted on the current term, the function returns nothing.

Suspend

This function is called by the client.

`Suspend(period)`

Makes the server sleep for `period` seconds.
Used to simulate a short-term disconnection of the server from the system.

Should print `Sleeping for <period> seconds`.

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 `getleader` and `suspend` commands will be now sent to this address.
2. `getleader` - requests the current Leader's id and the address from the server.
3. `suspend <period>` - makes the server to sleep for `<period>` seconds.
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:

`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: getleader
0 127.0.0.1:50000                 // This result is also sent to the client
Command from client: suspend 2
Sleeping for 2 seconds
I am a follower. Term: 2          // After receiving a heartbeat from a new Leader
Command from client: getleader
2 127.0.0.1:50002
```

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
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
The leader is dead                 // Actually, it is sleeping for 2 seconds
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 50000
>getleader
0 127.0.0.1:50000
```

```
>suspend 2
>getleader                      //after 2s
2 127.0.0.1:50002
>quit
The client ends
```

Hints

Visualization of the raft protocol:

http://thesecretlivesofdata.com/raft/

Possible implementation of a timer:

```
# busy waiting for <time> milliseconds
def wait_for(time):
  timing = datetime.datetime.now()
  while True:
    if (datetime.datetime.now() - timing).total_seconds() * 1000 > time:
        break
```

Be aware that your hardware limits the accuracy of the timers. Also, due to differences in operating systems, the timer accuracy in Windows is 16 ms, and Linux is 1 ms.

Server

You can divide your server in the following **threads** (so they share the same memory):

1. RPC server to accept requests and reset the timer.
2. RPC client to send requests.
3. Timer, which changes state if triggered. There should be a possibility to reset this timer from another thread (f.e. when a message is received).