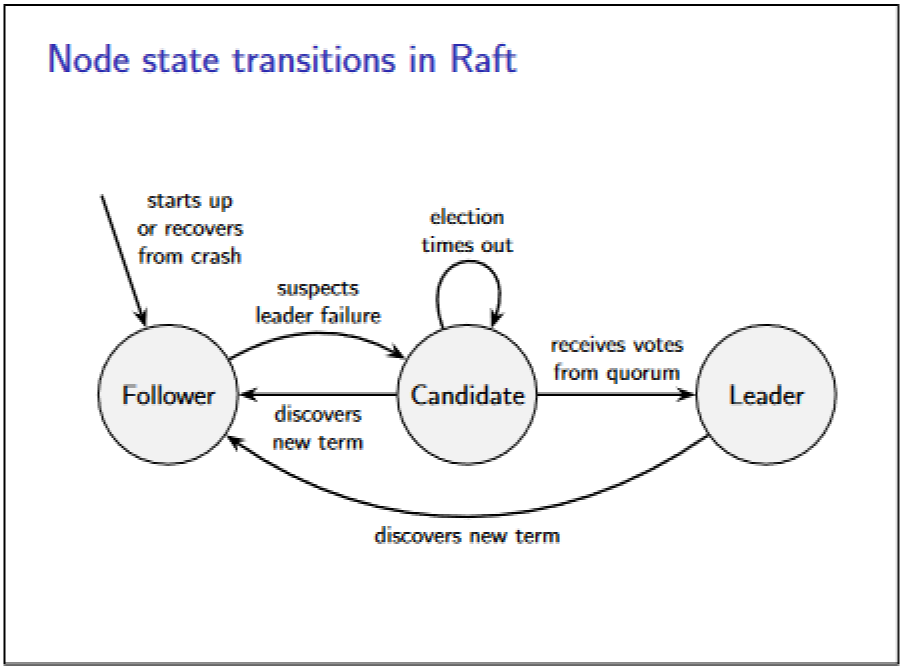
Assignment - 3

Raft Voting Algorithm

Authors : **MAB, MYM, MAA & MHH**

Version : **2 (13-03-2023)**

Problem Description



Source: *Distributed Systems* by Dr. Martin Kleppman

One of the algorithms that create consensus between nodes while being fault-tolerant is the raft algorithm. Taken from Distributed Systems by Dr. Martin Kleppman with slight modifications:

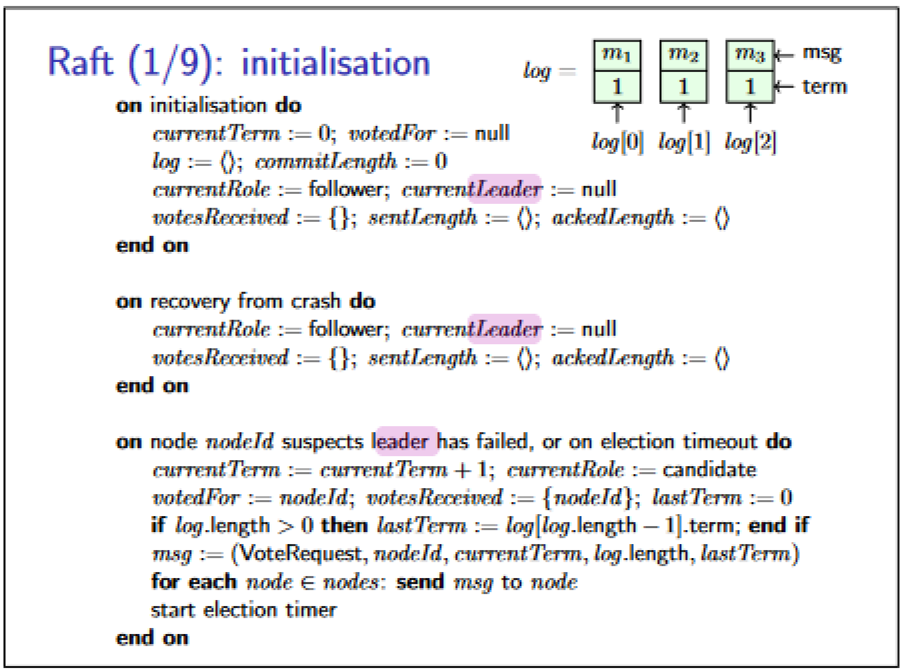
> The consensus problem is traditionally formulated as follows: several nodes want to come to agreement about a value. One or more nodes may propose a value, and then the consensus algorithm will decide on one of those values. The algorithm guarantees that the decided value is one of the proposed values, that all nodes decide on the same value (with the exception of faulty nodes, which may not decide anything), and that the decision is final (a node will not change its mind once it has decided a value).
>
> The two best-known consensus algorithms are Paxos and Raft. In its original formulation, Paxos provides only consensus on a single value, and the Multi-Paxos algorithm is a generalisation of Paxos that provides FIFO-total order broadcast. On the other hand, Raft is designed to provide FIFO-total order broadcasts “out of the box”.

> **At the core of most consensus algorithms is a process for electing a new leader when the existing leader becomes unavailable for whatever reason.**

In this assignment, you have to implement the leader election procedure in the raft algorithm. The procedure follows the pseudocode from *Distributed Systems* by Dr. Martin Kleppman which can be seen in the following section. You have to use Python with the socket programming concept to solve this problem; furthermore, correctness and understandability (e.g. easy to understand comments) will be evaluated. It should be noted that you have to complete the TODO section in `node.py` without making a change in `main.py`.

Leader Election in Raft Algorithm

The below screenshots are taken from *Distributed Systems* by Dr. Martin Kleppman



Raft (1/9): initialisation

```
on initialisation do
    currentTerm := 0; votedFor := null
    log := ⟨⟩; commitLength := 0
    currentRole := follower; currentLeader := null
    votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩
end on

on recovery from crash do
    currentRole := follower; currentLeader := null
    votesReceived := {}; sentLength := ⟨⟩; ackedLength := ⟨⟩
end on

on node nodeId suspects leader has failed, or on election timeout do
    currentTerm := currentTerm + 1; currentRole := candidate
    votedFor := nodeId; votesReceived := {nodeId}; lastTerm := 0
    if log.length > 0 then lastTerm := log[log.length − 1].term; end if
    msg := (VoteRequest, nodeId, currentTerm, log.length, lastTerm)
    for each node ∈ nodes: send msg to node
    start election timer
end on
```

Raft (2/9): voting on a new leader

c for candidate

```
on receiving (VoteRequest, cId, cTerm, cLogLength, cLogTerm)
        at node nodeId do
    if cTerm > currentTerm then
        currentTerm := cTerm; currentRole := follower
        votedFor := null
    end if
    lastTerm := 0
    if log.length > 0 then lastTerm := log[log.length − 1].term; end if
    logOk := (cLogTerm > lastTerm) ∨
            (cLogTerm = lastTerm ∧ cLogLength ≥ log.length)

    if cTerm = currentTerm ∧ logOk ∧ votedFor ∈ {cId, null} then
        votedFor := cId
        send (VoteResponse, nodeId, currentTerm, true) to node cId
    else
        send (VoteResponse, nodeId, currentTerm, false) to node cId
    end if
end on
```

Raft (3/9): collecting votes

```
on receiving (VoteResponse, voterId, term, granted) at nodeId do
    if currentRole = candidate ∧ term = currentTerm ∧ granted then
        votesReceived := votesReceived ∪ {voterId}
        if |votesReceived| ≥ ⌈(|nodes| + 1)/2⌉ then
            currentRole := leader; currentLeader := nodeId
            cancel election timer
            for each follower ∈ nodes \ {nodeId} do
                sentLength[follower] := log.length
                ackedLength[follower] := 0
                ReplicateLog(nodeId, follower)
            end for
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
        cancel election timer
    end if
end on
```

Raft (4/9): broadcasting messages

on request to broadcast msg at node $nodeId$ **do**
    if $currentRole = \text{leader}$ **then**
        append the record (msg : msg, term : $currentTerm$) to log
        $ackedLength[nodeId] := log.\text{length}$
        for each $follower \in nodes \setminus \{nodeId\}$ **do**
            $\textsc{ReplicateLog}(nodeId, follower)$
        end for
    else
        forward the request to $currentLeader$ via a FIFO link
    end if
end on

periodically at node $nodeId$ **do**
    if $currentRole = \text{leader}$ **then**
        for each $follower \in nodes \setminus \{nodeId\}$ **do**
            $\textsc{ReplicateLog}(nodeId, follower)$
        end for
    end if
end do

Raft (5/9): replicating from leader to followers

Called on the leader whenever there is a new message in the log, and also periodically. If there are no new messages, $suffix$ is the empty list. LogRequest messages with $suffix = \langle\rangle$ serve as heartbeats, letting followers know that the leader is still alive.

function $\textsc{ReplicateLog}(leaderId, followerId)$
    $prefixLen := sentLength[followerId]$
    $suffix := \langle log[prefixLen], log[prefixLen + 1], \ldots,$
            $log[log.\text{length} - 1]\rangle$
    $prefixTerm := 0$
    if $prefixLen > 0$ **then**
        $prefixTerm := log[prefixLen - 1].\text{term}$
    end if
    send (LogRequest, $leaderId, currentTerm, prefixLen,$
        $prefixTerm, commitLength, suffix$) to $followerId$
end function

Raft (6/9): followers receiving messages

```
on receiving (LogRequest, leaderId, term, prefixLen, prefixTerm,
              leaderCommit, suffix) at node nodeId do
    if term > currentTerm then
        currentTerm := term; votedFor := null
        cancel election timer
    end if
    if term = currentTerm then
        currentRole := follower; currentLeader := leaderId
    end if
    logOk := (log.length ≥ prefixLen) ∧
             (prefixLen = 0 ∨ log[prefixLen − 1].term = prefixTerm)
    if term = currentTerm ∧ logOk then
        AppendEntries(prefixLen, leaderCommit, suffix)
        ack := prefixLen + suffix.length
        send (LogResponse, nodeId, currentTerm, ack, true) to leaderId
    else
        send (LogResponse, nodeId, currentTerm, 0, false) to leaderId
    end if
end on
```

How to Run the Program

To run the program, you can execute

```
python main.py
```

Please run

```
python main.py -h
```

To see the explanation of the arguments if you want to use them.

There are also inputs that you can give to the program

1. `k<node_id>` for killing a node. For example, `k1` kills Node 1
2. `r<node_id>` for restarting a node.
3. `e` for killing all the nodes

Important Notes

1. There are other sources that explain raft algorithm with visualization such as
    a. https://raft.github.io/
    b. http://thesecretlivesofdata.com/raft/
2. You only need to implement the leader election procedure and not the total broadcast procedure of raft algorithm
3. This program runs on Python 3.8.8
4. It is encouraged to use the logging technique to debug this distributed system.

Submission

There is only one file that you have to submit in this assignment:

1. `node.py`

Zip that file and write the filename as `A3-YourName.zip`. For example, `A3-MuhammadAriqBasyar.zip`.

Troubleshooting

If you have a question regarding this assignment, you can post your question on this assignment week's discussion board **without posting any of your codes**. With this in mind, you could post a default code on the discussion board. Anyone can reply to the question.

Scoring Criteria

There are three components that will be evaluated, with two having sub-components.

1. Cases: 30%
    a. All nodes are running: 10%
    b. The first leader node is stopped and restarted: 10%
    c. The first and the second leader nodes are stopped and restarted: 10%
2. Election timer procedure: 10%
3. Pseudocode from slides above: 60%
    a. Each pseudocode of a slide weighs 10%

The weight of each scoring criterion can be seen below:

1. Correctness: 90%
2. Understandability: 10%

The procedure to evaluate this assignment is by demonstrating the program to the teaching assistant.

Penalty

The rule for a late submission with X is the amount of time after the deadline:

- X < 10 minutes : 0%
- 10 minutes ≤ X < 2 hours : 25%
- 2 hours ≤ X < 4 hours : 50%
- 4 hours ≤ X < 6 hours : 75%
- X ≥ 6 hours : **REJECTED**