MP2: Raft Implementation

Due Date: CP1: Wednesday, Nov 10, 11:59 p.m.

Due Date: CP2: Friday, Nov 19, 11:59 p.m.

Your goal for this MP will be to implement the [Raft consensus algorithm][https://raft.github.io/]. You will be implementing the leader election and logging components of the algorithm, including deadling with node failures and partitions. You will not need to implement crash recovery, log compaction, or cluster membership changes. Unlike in the previous MPs, you will use a simulation framework to communicate between processes by writing them to and reading them from `stdout` / `stdin`. The framework allows us to simulate things like partitions, message delays, and process crashes, and to create test harnesses. You will be provided with a set of tests that your code should pass. Passing the tests will be sufficient for getting the credit for this assignment; you will not need to do measurement or design for this MP.

Messaging

There are *n* nodes in the cluster, with identities 0, …, *n*-1. Each node will be given its own identity and the total number of nodes in the cluster, *n*. To send messages to the other nodes, the node should write a message to `stdout`: "`SEND` *k* <*message*>" followed by a newline (`\n`). For example, node 0 might send a RequestVotes message:

```
SEND 3 RequestVotes 2
```

The simulation framework will then take the message and deliver it to the correct node in the cluster. The message will be delivered in a similar format, except that the second entry will be the *source* of the message:

```
RECEIVE 0 RequestVotes 2
```

The framework will copy the contents of the message verbatim after the second space character, so you may include any information in the message. It may include spaces but it may not include a newline. Using, e.g., a JSON library can save you time implementing the parsing and formatting of messages. For debugging purposes a text format is recommended, but if you choose a binary representation be sure to encode it using something like base64.

State updates

The test harness will need to see some internal state of each node to make sure that Raft invariants are being followed. To do this, you need to output (to `stdout`) a message "`STATE` *var*=*value*". For example:

```
STATE term=3
STATE state="FOLLOWER"
STATE leader=0
STATE log[2]=[3,"hurwity"]
STATE commitIndex=2
```

The variables you need to track are:

- *term* — The current term according to the node
- *state* — One of `LEADER`, `FOLLOWER`, or `CANDIDATE` (please use all caps)
- *leader* — The identity of the the leader of the current term, if known
- *log*`[`*i*`]` — The term and contents of the log entry *i*, including the term number and the contents. In the example above, log entry #2 has term 3 and contents "hurwity". (Log entries are numbered starting at 1.)
- *commitIndex* — The index of the last committed entry.

Logging messages

The framework will give you entries to log. The entries will look like "`LOG` <*string*>", where the string is a base64-encoded unique identifier. The string should be added to the log and committed. Both the leaders and the followers should add the entry to their logs (as reported by a `STATE log[...]` message). Once the message is committed, the leader should acknowledge the commit and its log position "`COMMITTED` <*string*> *k*". (This should not be done until the message has been replicated, of course.)

If a node receives a `LOG` message while not in a LEADER state, it is allowed to ignore it. (Real Raft would reply to the message with an error redirecting the request to the leader.) Also, if a term changes before the leader has a chance to fully commit a message, it is no longer required to send a response, whether the message is committed or not.

Running the node

To run the node, you should have an executable called `raft` in the base directory of your submission. If you are using a compiled language, such as Go or C++, you **MUST** include a `Makefile` that compiles the executable. Otherwise, you can make it a shell script, such as:

```
python3 raft.py $@
```

Remember to make the script executable (`chmod +x raft`). Your executable will take 2 arguments: the identity of the current raft node and the number of nodes; e.g., `raft 3 5`.

Getting Started

The framework code is published on GitHub; you can download the latest version at https://github.com/nikitaborisov/raft_mp:

```
git clone https://github.com/nikitaborisov/raft_mp.git
```

To run the framework, you will need a recent version of Python. You can install Python3.10 on a VM by running:

```
cd
wget https://courses.engr.illinois.edu/cs425/fa2021/assets/python3.10.tar.gz
cd /usr/local
sudo tar xzvf ~/python3.10.tar.gz
cd
```

After this, you should be able to run the pinger code:

```
cd ~/raft_mp
python3.10 framework.py 3 python3.10 pinger.py
```

If you want to log messages, you will need to install the `aioconsole` module:

```
pip3.10 install aioconsole
```

Checkpoints

Checkpoint one (CP1) has to implement the leader election component only, without the logging component. The second checkpoint (CP2) should implement full functionality. We will retest your CP1 functionality in your CP2 submission so you will get some points for implementing it if you do not get it finished fro CP2. (Conversely, make sure your implementation of CP2 does not regress in CP1 functionality!)

Your points will be assessed based on passing the tests. Note that the tests are not deterministic, so it is possible that you can pass it once in your VM environment and nevertheless encounter an error during grading.

Grade Breakdown

- CP1: 30 points, tested with up to 5 nodes
  - Initial election: 10 points
  - New election after leader failure: 10 points
  - Partition test: 10 points
- CP2: 70 points, tested with up to 9 nodes
  - Retest CP1: 15 points
  - Simple log test: 11 points
  - Log 5 test: 11 points
  - Follower failure test: 11 points
  - Leader failure test: 11 points
  - Log partition test: 11 points

Sample Transcript

In the transcript below there are 3 nodes, 0, 1, and 2. Lines sent by the framework to process *i*'s stdin are labeled with `i<`, lines written to stdout are labeled with `i>`. The message contents here has been simplified from a full Raft implementation.

```
0> STATE state="CANDIDATE"
0> STATE term=2
0> SEND 1 RequestVotes 2
0> SEND 2 RequestVotes 2
1< RECEIVE 0 RequestVotes 2
1> SEND 0 RequestVotesResponse 2 true
2< RECEIVE 0 RequestVotes 2
2> SEND 0 RequestVotesResponse 2 true
0< RECEIVE 1 RequestVotesResponse 2 true
0> STATE state="LEADER"
0> STATE leader=0
0> SEND 1 AppendEntries 2 0
0> SEND 2 AppendEntries 2 0
0< RECEIVE 2 RequestVotesResponse 2 true
1< RECEIVE 0 AppendEntries 2 0
1> STATE term=2
1> STATE state="FOLLOWER"
1> STATE leader=0
1> SEND 0 AppendEntriesResponse 2 true
2< RECEIVE 0 AppendEntries 2 0
2> STATE term=2
2> STATE state="FOLLOWER"
2> STATE leader=0
2> SEND 0 AppendEntriesResponse 2 true
0< RECEIVE 1 AppendEntriesResponse 2 true
0< RECEIVE 2 AppendEntriesResponse 2 true
0< LOG ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw
0> STATE log[1]=[2,"ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
0> SEND 1 AppendEntries 2 0 ["ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
0> SEND 2 AppendEntries 2 0 ["ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
1< RECEIVE 0 AppendEntries 2 0 ["ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
1> STATE log[1]=[2,"ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
1> SEND 0 AppendEntriesResponse 2 true
2< RECEIVE 0 AppendEntries 2 0 ["ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
2> STATE log[1]=[2,"ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw"]
2> SEND 0 AppendEntriesResponse 2 true
0< RECEIVE 1 AppendEntriesResponse 2 true
0> STATE commitIndex=1
0> COMMITTED ru9pOYHk1xHxVRaMhc7k5N1uPNn_ryOtPfv6ZU__5Aw 1
[...]
```