April 14, 2022

ECE428 Machine Programming 2

Due: 11:59 p.m. on Monday, May 23rd, 2022

This MP is about implementing the Raft consensus following its description at `raft.github.io`. The MP is optional, but can be rewarded up to 5 marks, if submitted by one or a group of two students, and 2.5 (or 2) marks, if submitted by a group of 3 (or 4) students. The student groups can be different from MP0 and MP1. Any programming language can be used, however, please contact the TA first, if you want to use a language other than C/C++, Java or Python. You can reuse and extend your code from MP0 and MP1.

In particular, you will be implementing a leader election and also logging the steps of Raft algorithm, and be expected to deal with node failures and partitions, but you do not need to implement crash recovery, log compaction, and cluster membership changes. However, unlike in previous MPs, you will use a framework to simulate communications between processes by writing to stdout and reading from stdin. This framework allows us to simulate things such as partitions, message delays, and process crashes, and also 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; you will not need to collect measurements for this MP.

Messaging

There are *n* nodes in the cluster, with IDs 0, …, *n*-1. Each node is given its own unique identity and is aware of the total number of nodes, *n*, in the cluster. In order to send messages to 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 get the message from stdout, and delivers it to a correct node in the cluster. The message is delivered in a similar format, except that the second entry is the *source* of the message:

```
RECEIVE 0 RequestVotes 2
```

The communication framework copies the contents of messages verbatim after the second space character, so any information can be included in the message. The information may contain spaces, but it may not contain a newline. Using, for example, a JSON library can save time in implementing parsing and formatting the messages. For debugging purposes, a simple text format is recommended; if you choose a binary representation, make sure to encode it using something like base64.

State updates

The test harness will need to access some internal states at each node to make sure that Raft invariants are being followed. For this, you will need to output (to stdout) a message formatted as: "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 leader in the current term (if known).
- *log*[*i*] — The term number and contents of the log entry *i*. In the example above, log entry #2 has term 3 and the content, "hurwity". (Log entries are numbered starting at 1.)
- *commitIndex* — The index of the last committed entry.

Logging messages

The communication framework will give you entries to the 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 the message, it is no longer required to send a response, even if the message has not been yet committed.

Running nodes

To run a node, you should have an executable called raft in the base directory of your submission. If you are using a compiled language (e.g., C/C++, Java), 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 code of the communication framework is published on GitHub, and you can download the latest version at https://github.com/nikitaborisov/raft_mp using the command:

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

The framework requires a more recent version of Python (3.10 or newer):

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

In order to log messages, please install the aioconsole module:

```
pip3.10 install aioconsole
```

Checkpoints

Checkpoint one (CP1) has to only implement the leader election component without the logging component. The second checkpoint (CP2) should implement a full functionality. We will retest your CP1 functionality within your CP2 submission so you will get some points for implementing it, even if you do not completely achieve full functionality in CP2. (Thus, 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 environment and still encounter an error during grading. (To minimize chances of this happening, rerun the test multiple times in your environment.)

CP1: tested with up to 5 nodes

- Initial election
- New election after leader failure
- Partition test

CP2: tested with up to 9 nodes

- Log test
- Follower failure test
- Leader failure test

Sample transcript

In the transcript below, there are 3 nodes with IDs 0, 1, and 2. The lines sent by the framework to process *i*'s stdin are labeled with i<, the lines written to stdout are labeled with i>. The message contents have 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
[...]
```