Lab Experiment 4: The Raft Consensus Algorithm

In this week's experiment you will learn:
1. Raft and installing raftos
2. Run a raftos cluster and demonstrate leader election.
3. Verify that data is being logged/stored by all nodes involved

What is Consensus?

To quote a *very wise man*: "Sometimes, when two (or more) computers love each other very much, they talk to each other over a network; and 1-RTT later, a very cute distributed system is born :3". For more details, check rowjee.com/blog/papers/raft!

A fundamental problem in distributed computing is to achieve **overall system reliability** in the presence of a **number of faulty processes.** This is a natural consequence of life, because machines fail. Power cords disconnect. Processes error out. Tsunamis happen. A vengeful employee burns down a datacenter.

This often requires coordinating processes to reach **consensus**, or agree on some data value that is needed during computation. Example applications of consensus include agreeing on what transactions to commit to a database in which order, **state machine replication**, and atomic broadcasts.

Consider a system of **five servers**, each of which is responsible for performing some computation.
1. A client sends out a message that says
Set COOLEST_HACKATHON = 'somerandomhackathon'; this is registered by all five.

2. Later, the client sends another message saying
Set COOLEST_HACKATHON = 'HashCode'

Except, before the client sent this**, one of the machines failed;**
Later it came back online, **after** the message was received by everyone else. Now if that specific machine is queried for 'COOLEST_HACKATHON', it would get a reply saying '*somerandomhackathon*', which is just obviously not true.

Real-world applications often requiring consensus include cloud computing, clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain, and others.

What is Raft?

You've learnt about Paxos in class. Raft is another consensus algorithm, just like Paxos, except simpler to understand and implement. Here's what the Raft website has to say: "Raft is a consensus algorithm that is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems. We hope Raft will make consensus available to a wider audience, and that this wider audience will be able to develop a variety of higher quality consensus-based systems than are available today."

Kubernetes' etcd is implemented using Raft!

Prerequisites:

1. **Pyenv/Conda**
2. **Tmux**
3. **Linux/Linux-VM/macOS system. If you HAVE to use windows, use WSL2.**

The submission consists of **two components:** The screenshots, and the **zip file of the logs generated.**

A. The following screenshots are to be submitted in a PDF file:
   a. 1a: Pyenv installed and local Python version changed to 3.6.8
   b. 2a: Initial leader election has occurred, and first value has replicated.
   c. 2b: Leader crashed, new leader elected, and second value replicated.
   d. 2c: Old leader comes back online, but becomes a follower instead and updates its old state to its new state.

Please make sure the folder name (i.e, CC_E4_<YOUR_SRN>) is clearly visible in the current working directory in the screenshots.

Name the file **<SECTION>_<SRN>_<NAME>_E4** (eg: *E_PES1UG19CS999_Rumali_E4.pdf*").

B. In addition, all the .log (including CUSTOMLOG), .state_machine and .storage files need to be zipped, with the zip file named the same as above, and submitted.

Instructions:

Please read **ALL the instructions** carefully before proceeding.

Task 1: Raft and installing Raftos

1. Go through this visualization (http://thesecretlivesofdata.com/raft/) to understand how raft works. After completing the above visualization, you should be familiar with leader election and log replication, and how they work in **Raft**. You will be quizzed on basic Raft concepts! If you're interested in actually **interacting** and playing around with a raft-setup, https://raft.github.io/ is your friend.

2. Navigate into the raftos directory **that you downloaded in the pre-install section,** and activate the local python version as 3.6.8 using **pyenv.** If you don't know what this means, go through the **preinstall guide.**
   Your terminal should then look something like this: (Take a screenshot, **1a**)

   ```
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ ls -a
   .  ..  raftos
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ python -V
   Python 3.10.6
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ pyenv versions
   * system (set by /home/laruim/.pyenv/version)
     3.6.8
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ pyenv local 3.6.8
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ python -V
   Python 3.6.8
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ ls -a
   .  ..  .python-version  raftos
   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000$ []
   ```

   If you're using Conda, you'll need to do `conda activate py3.8.11`. Take a screenshot with the environment version clearly visible.

3. Navigate into the raftos directory and edit the requirements.txt file by changing the version of the *cryptography* library from **1.5.3** to **3.3,** because stuff has changed.

4. Now run `pip3 install setuptools` and then run `pip3 install -r requirements.txt`

5. Now, run `python3 setup.py install` in raftos after you're done correcting the version in requirements.txt. This should install raftos on your system. You may have to sudo the command if it errors out.

Task 2: Seeing Raft in Action

1. We're going to be writing a Python script that simulates the running of a basic server. Each *instance* of the script corresponds to one *node*, i.e one *server* that is part of the **cluster.** The only thing differentiating them will be their **node-id's**.
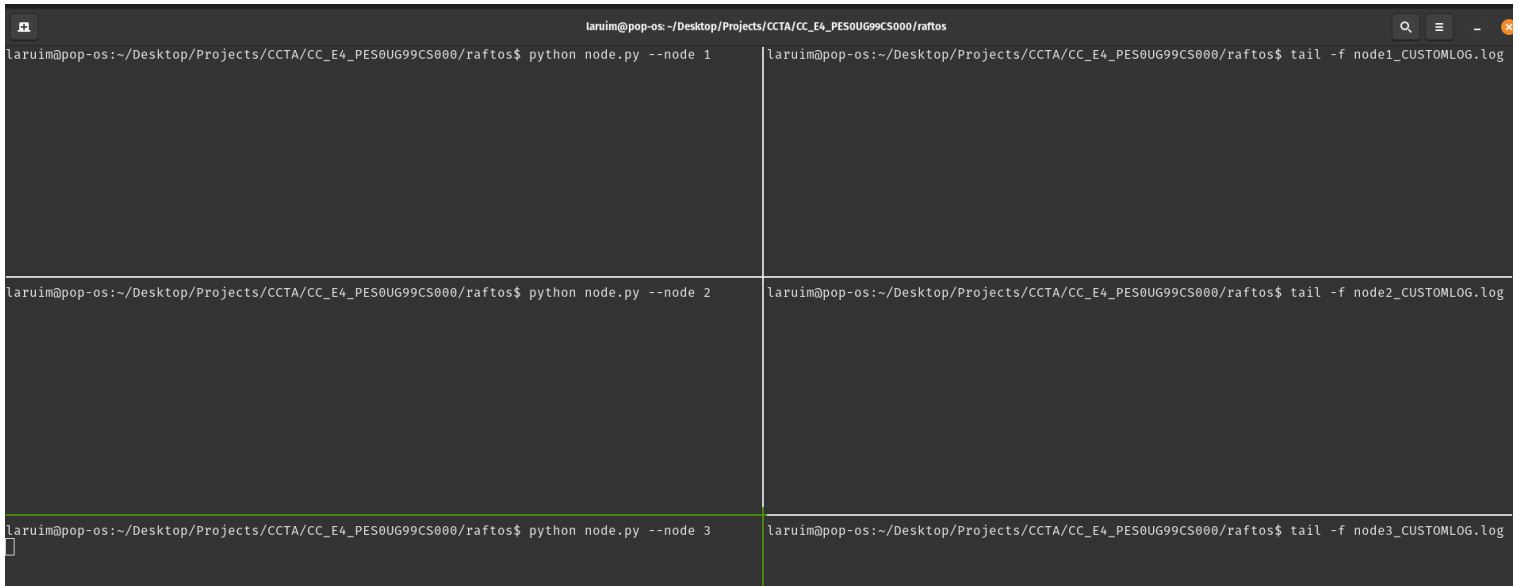
    Each node maintains three descriptors; a **.log file** that corresponds to *command logs*; a **.storage** file that corresponds to the node's *persistent storage*; and a **.state_machine** file that corresponds to the node's *state.* Simply put (and this is a bit of an oversimplification), the '**command log**' contains the set of **instructions** sent by the client; on **applying** that instruction, the **state** of the node changes.

    For example, a command in the command log might be "Set X to 4"
    And, that, on application, will change the **state** of X to 4.

    Why do we log the commands too, instead of just applying? One intuitive reason (of many) is for **failure recovery**. If a node fails and comes back, the state of the node may end up being wiped; Applying these logs sequentially will bring the node back to its former state.

2. Since the main goal of this experiment is to **understand** Raft, the code has been provided for you already; Place the **node.py** file provided to you in the raftos directory. Also place the **start_terminals.sh** file provided to you in this directory.

3. Replace 'somerandomhack' in node.py with **your SRN. This is important for evaluation.**

4. **Before executing any runs of the experiment, make sure to run start_terminals.sh.**
    You may need to provide it with executable permissions, with `sudo chmod +x ./start_terminals.sh`

5. This should open up **6 tmux panes in that terminal.** Don't fret (I don't ever wanna see you And I never wanna miss you again), you can easily navigate between each pane by first pressing Ctrl+b (or if that doesn't work, cmd+b for Mac users) and using the arrow keys to change panes. Each time you need to change panes, you'll need to ctrl+b, unless you do it in quick succession. **Do not press ctrl+b+arrow key together**, instead press the arrow key a second after the ctrl+b. Pressing them together **resizes** the window.
    If you're using Conda, you'll need to do `conda activate py3.8.11` in **every tmux pane. (yes, again)**

6. **Make your tmux session full screen**.

7. Run `tail -f node1_CUSTOMLOG.log`, `tail -f node2_CUSTOMLOG.log` and `tail -f node3_CUSTOMLOG.log` in three panes in the right hand side column, in that order from top to bottom.

8. Run `python node.py --node 1`, `python node.py --node 2` and `python node.py --node 3` in the three panes in the first columns, in that order from top to bottom. Now, your tmux session should look something like this:
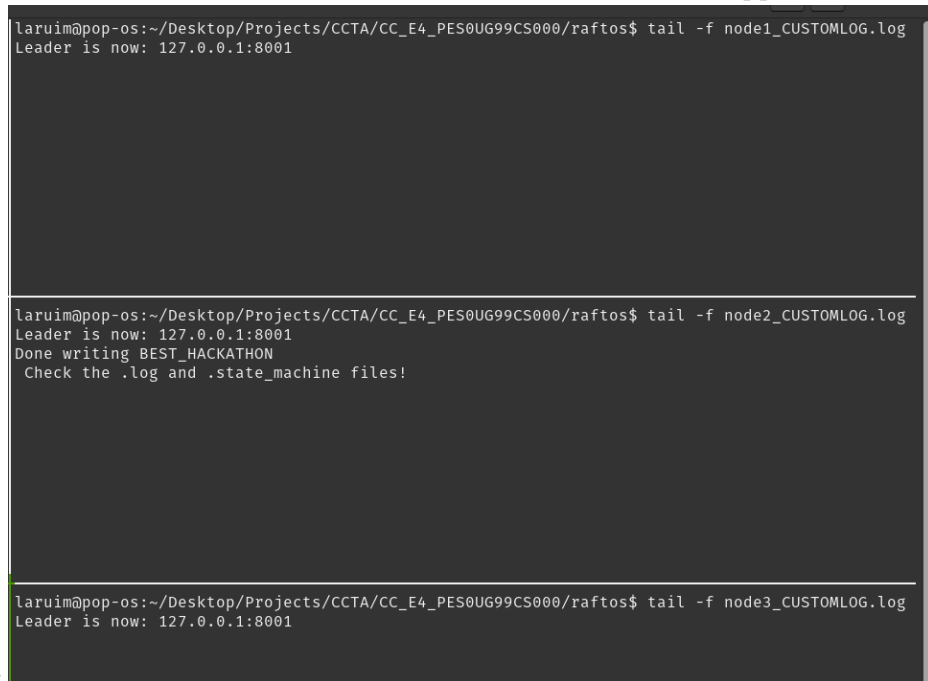
Soon, leader election will happen, post which one of the nodes will be elected leader, and will try to set the value of a variable to a certain value.

9. In my case, Node 2, i.e 127.0.0.1:8001, was elected leader, and hence this happens: (Take a

screenshot, **2a**):

Please make sure the folder name (i.e, CC_E4_<YOUR_SRN>) is clearly visible in the current working directory in all screenshots.

10. We'll do as suggested and check the .log and .state_machine files:
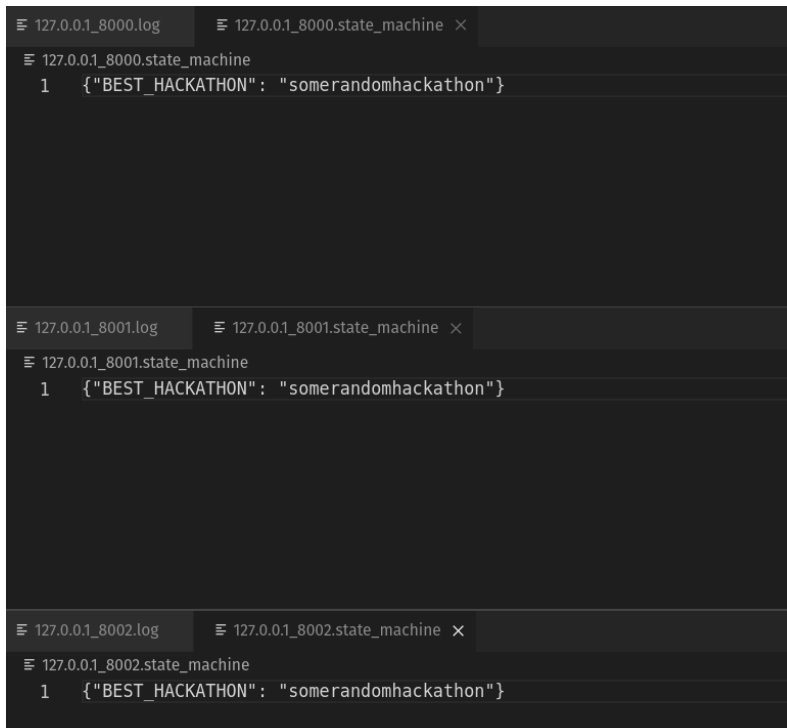
```
≡ 127.0.0.1_8000.log ×
  ≡ 127.0.0.1_8000.log
    1    {"term": 1, "command": {"BEST_HACKATHON": "somerandomhackathon"}}
    2

≡ 127.0.0.1_8001.log ×
  ≡ 127.0.0.1_8001.log
    1    {"term": 1, "command": {"BEST_HACKATHON": "somerandomhackathon"}}
    2    |

≡ 127.0.0.1_8002.log ×
  ≡ 127.0.0.1_8002.log
    1    {"term": 1, "command": {"BEST_HACKATHON": "somerandomhackathon"}}
    2
```

It's alright if your term varies! That's just an artifact of the election process itself, which you'll hopefully have understood via the visualisation. **(You don't need to show these screenshots)**

```
≡ 127.0.0.1_8000.log        ≡ 127.0.0.1_8000.state_machine ×
  ≡ 127.0.0.1_8000.state_machine
    1    {"BEST_HACKATHON": "somerandomhackathon"}

≡ 127.0.0.1_8001.log        ≡ 127.0.0.1_8001.state_machine ×
  ≡ 127.0.0.1_8001.state_machine
    1    {"BEST_HACKATHON": "somerandomhackathon"}

≡ 127.0.0.1_8002.log        ≡ 127.0.0.1_8002.state_machine ×
  ≡ 127.0.0.1_8002.state_machine
    1    {"BEST_HACKATHON": "somerandomhackathon"}
```

11. We see now that the initial value has been replicated across all the nodes. Now it's time to demonstrate leader election. Go back to your tmux session, and stop the leader process with ctrl+c. This simulates leader failure. Wait, and you should see a new leader elected, and the second value replicated across the cluster! In **my** case, the old leader 's **node 2 (but yours may be different)**, so it'll look something like this: (Take a screenshot, **2b**)

```
laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 1        laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node1_CUSTOMLOG.log
                                                                                                  Leader is now: 127.0.0.1:8001
                                                                                                  Leader is now: 127.0.0.1:8000
                                                                                                  Done writing BEST_HACKATHON
                                                                                                   Check the .log and .state_machine files!




laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 2        laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node2_CUSTOMLOG.log
^CTraceback (most recent call last):                                                              Leader is now: 127.0.0.1:8001
  File "node.py", line 85, in <module>                                                            Done writing BEST_HACKATHON
    loop.run_forever()                                                                             Check the .log and .state_machine files!
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/asyncio/base_events.py", line 438, in
run_forever
    self._run_once()
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/asyncio/base_events.py", line 1415, in
_run_once
    event_list = self._selector.select(timeout)
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/selectors.py", line 445, in select
    fd_event_list = self._epoll.poll(timeout, max_ev)
KeyboardInterrupt
laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$
laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 3        laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node3_CUSTOMLOG.log
                                                                                                  Leader is now: 127.0.0.1:8001
                                                                                                  Leader is now: 127.0.0.1:8000
```

12. As before, **check your .state_machine file and your .log file**; Ta da! New values have been replicated across the nodes that are still up.

13. The best part is, if you were to revive the old leader (**for me, node 2**) now, by starting the process again, it'll notice that **its** term is lesser than the current leader's term (**for me, node 1**), become a follower, and also update its state to reflect the new value! (Take a screenshot, **2c**)

```
laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 1        laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node1_CUSTOMLOG.log
                                                                                                  Leader is now: 127.0.0.1:8000
                                                                                                  Done writing EXAMPLE_VAR
                                                                                                   Check the .log and .state_machine files!




  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/asyncio/base_events.py", line 471, in   laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node2_CUSTOMLOG.log
run_until_complete                                                                                Leader is now: 127.0.0.1:8000
    self.run_forever()                                                                            tail: node2_CUSTOMLOG.log: file truncated
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/asyncio/base_events.py", line 438, in    Leader is now: 127.0.0.1:8000
run_forever
    self._run_once()
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/asyncio/base_events.py", line 1415, in
_run_once
    event_list = self._selector.select(timeout)
  File "/home/laruim/.pyenv/versions/3.6.8/lib/python3.6/selectors.py", line 445, in select
    fd_event_list = self._epoll.poll(timeout, max_ev)
KeyboardInterrupt
laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 2

laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ python node.py --node 3        laruim@pop-os:~/Desktop/Projects/CCTA/CC_E4_PES0UG99CS000/raftos$ tail -f node3_CUSTOMLOG.log
                                                                                                  Leader is now: 127.0.0.1:8000
```

FAQ and Common Issues

1. **.storage or .log files regenerate by themselves:**
   You closed the tmux session without ending the processes. You'll need to end them manually. For that, do:
   `ps -ef | grep python`

   Note down the process ids. Then, for each process number, do:
   `kill -9 <process_id>`

2. **Python version is not switching to 3.6.8:**
   You likely forgot to add certain lines to the .bashrc or .zshrc file; check the pre-install guide again.

3. **Other installation errors:**
   PES wi-fi may have blocked certain PPAs and repos; try switching to hotspot.