Instruction Manual - Consensus Algorithms: Raft

In this week's CC Lab, you will learn:

Task 1) Raft and installing raftos
Task 2) Run a raftos cluster and demonstrate leader election.
Task 3) Verify that data is being logged/stored by all nodes involved.

What is Consensus:?

A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a number of faulty processes. 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. 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[1].

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." [2]

Why Raft?

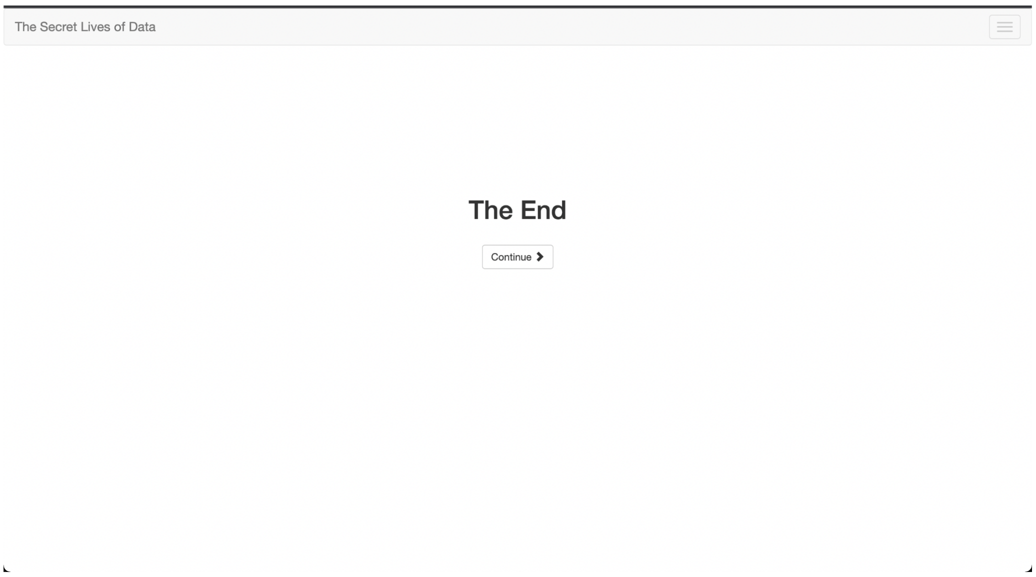
Raft was created by Diego Ongaro and John Ousterhout at Stanford University in 2014. It was created as an alternative to Paxos that is easier to understand and implement. Raft serves as a good introduction to consensus algorithms not just for its academic use but also because it has been implemented and used in a lot of tools, for example, etcd(Kubernetes' data store) and kafka(KIP500 implements KRaft)

Prerequisites:

- Python 3.9.x or below.
- Linux/macOS system/VM/container. If you ***HAVE*** to use windows, use WSL2(Windows Subsystem for Linux).

Task 1: Raft and installing Raftos

Step 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. Take a screenshot at the end of the visualization(1-a.jpg)



Step 2) Create a directory called ` YOUR_SRN_CCLAB_X` and clone the repo https://github.com/zhebrak/raftos inside it. Raftos is one of the many implementations of raft that are publicly available.

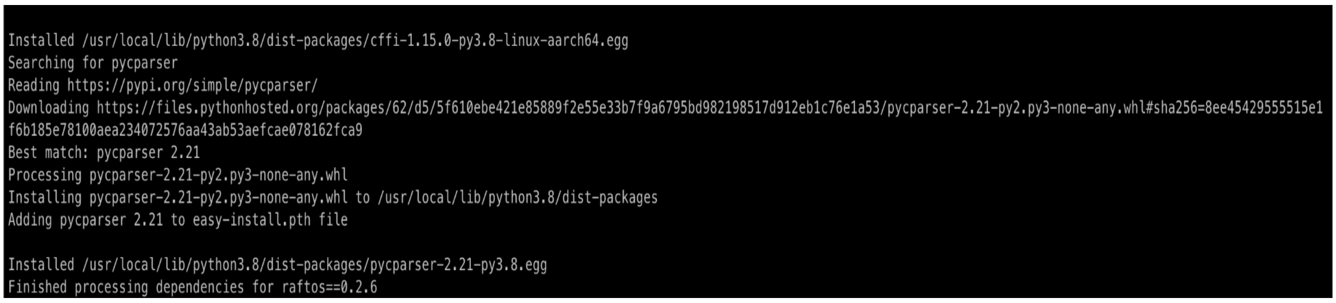
Step 3) Change the directory to raftos and edit the requirements.txt file inside it.

Step 4) Change the version of cryptography from `1.5.3` to `3.3` that is, line 1 of the file should be cryptography==3.3.

Step 5) Install python and pip and add them to your path. If you already have python installed then go to step 6.

Step 6) Run `pip install setuptools` to install setuptools.

Step 7) Run `python3 setup.py install` in raftos after you're done correcting the version in requirements.txt. This should install raftos on your system. Take a screenshot(1-b.jpg)

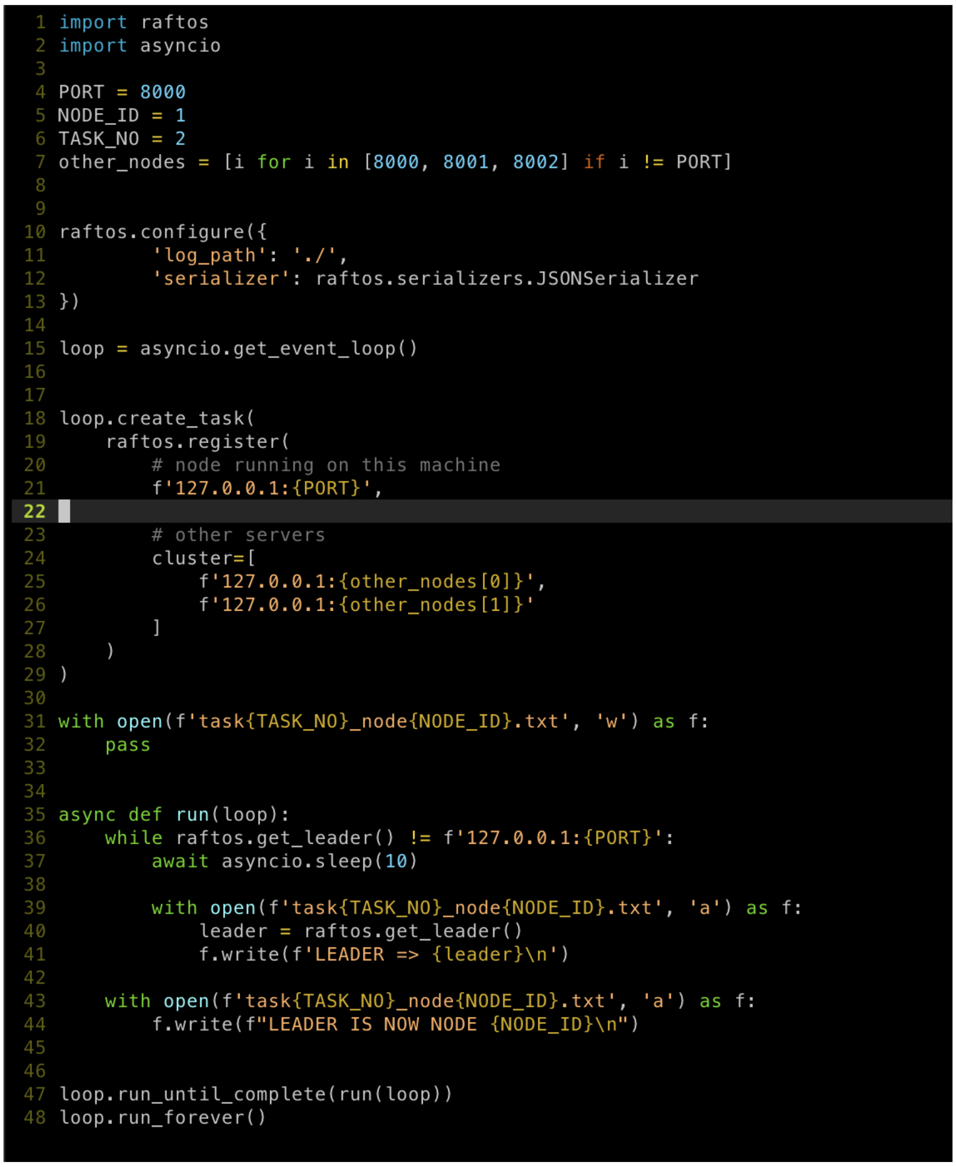


```
Installed /usr/local/lib/python3.8/dist-packages/cffi-1.15.0-py3.8-linux-aarch64.egg
Searching for pycparser
Reading https://pypi.org/simple/pycparser/
Downloading https://files.pythonhosted.org/packages/62/d5/5f610ebe421e85889f2e55e33b7f9a6795bd982198517d912eb1c76e1a53/pycparser-2.21-py2.py3-none-any.whl#sha256=8ee45429555515e1
f6b185e78100aea234072576aa43ab53aefcae078162fca9
Best match: pycparser 2.21
Processing pycparser-2.21-py2.py3-none-any.whl
Installing pycparser-2.21-py2.py3-none-any.whl to /usr/local/lib/python3.8/dist-packages
Adding pycparser 2.21 to easy-install.pth file

Installed /usr/local/lib/python3.8/dist-packages/pycparser-2.21-py3.8.egg
Finished processing dependencies for raftos==0.2.6
```

Task 2: Run a raftos cluster and demonstrate leader election

Step 1: Copy the following code into a file in `YOUR_SRN/` called 1.py.

Please write the f.write() statements as is, for evaluation purposes.



```python
import raftos
import asyncio

PORT = 8000
NODE_ID = 1
TASK_NO = 2
other_nodes = [i for i in [8000, 8001, 8002] if i != PORT]


raftos.configure({
        'log_path': './',
        'serializer': raftos.serializers.JSONSerializer
})

loop = asyncio.get_event_loop()


loop.create_task(
    raftos.register(
        # node running on this machine
        f'127.0.0.1:{PORT}',

        # other servers
        cluster=[
            f'127.0.0.1:{other_nodes[0]}',
            f'127.0.0.1:{other_nodes[1]}'
        ]
    )
)

with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'w') as f:
    pass


async def run(loop):
    while raftos.get_leader() != f'127.0.0.1:{PORT}':
        await asyncio.sleep(10)

        with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'a') as f:
            leader = raftos.get_leader()
            f.write(f'LEADER => {leader}\n')

    with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'a') as f:
        f.write(f"LEADER IS NOW NODE {NODE_ID}\n")


loop.run_until_complete(run(loop))
loop.run_forever()
```

Step 2) Notice how 1.py is running on PORT 8000 and the NODE_ID is 1, and the cluster contains nodes running on ports 8001 and 8002 as well. Notice that the TASK_NO is 2.

Step 3) Create 2 new copies of 1.py and call them 2.py and 3.py. Change the PORT number in 2.py from 8000 to 8001 and the NODE_ID from 1 to 2.

Step 4) Do the same thing with 3.py as well, except change the PORT from 8000 to 8002 and NODE_ID from 1 to 3.

Step 5) Run 1.py, 2.py, and 3.py in new terminal tabs and check if there are any errors. There should be none at this point, ideally. The point of this is to simulate a cluster with 3 nodes running.

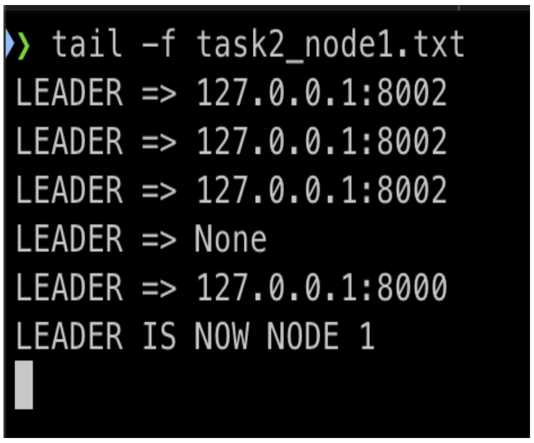
Step 6) Once you have all 3 files running, open new terminal tabs and run `tail -f task2_node{NODE_ID}.txt` to check the logs of each of these nodes. Do this for all 3 nodes.

Step 7) Once you have "LEADER IS NOW NODE X" in one of the node's logs(eg: node 2, that is, "LEADER IS NOW NODE 2 comes in the output of `tail -f task2_node2.txt`), press `Ctrl+c` in that node(eg: in the tab where 2.py is running, not in the tab where `tail` is running) to simulate the crashing of that node. Take a screenshot[2_a.jpg]. An example screenshot is given below(the image on the left)

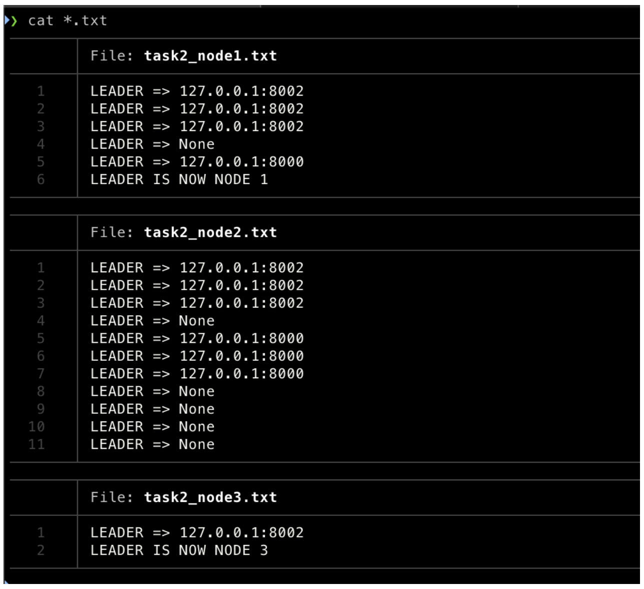
Step 8) In the logs of one of the two running nodes, you should now get a "LEADER IS NOW NODE Y". Press `Ctrl+C` here to once again simulate node failure. Take a screenshot[2_b.jpg]. Look at the contents of task2_node1.txt in the below example images for reference.

Step 9) Wait for the final node to print "LEADER => NONE" in its log a few times, just to confirm that no leader is going to be elected, as a candidate needs 2 votes to become a leader in a cluster of 3 nodes, but here we only have 1 node running. Look at the contents of the log file in the below example images for reference(node 2 in the image on the right). Take a screenshot[2_c.jpg].

The below images show what the contents of the log files looked like after a trial run. It is okay if the order of leaders is switched around or if the messages aren't in the same order. You just need to show that the leaders are being re-elected when there's a crash-stop failure, and that no leaders can be elected when there's only 1 node left in a cluster of 3 nodes(look at the contents of task2_node2.txt in the pictures below)



```
❯ tail -f task2_node1.txt
LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => None
LEADER => 127.0.0.1:8000
LEADER IS NOW NODE 1
```

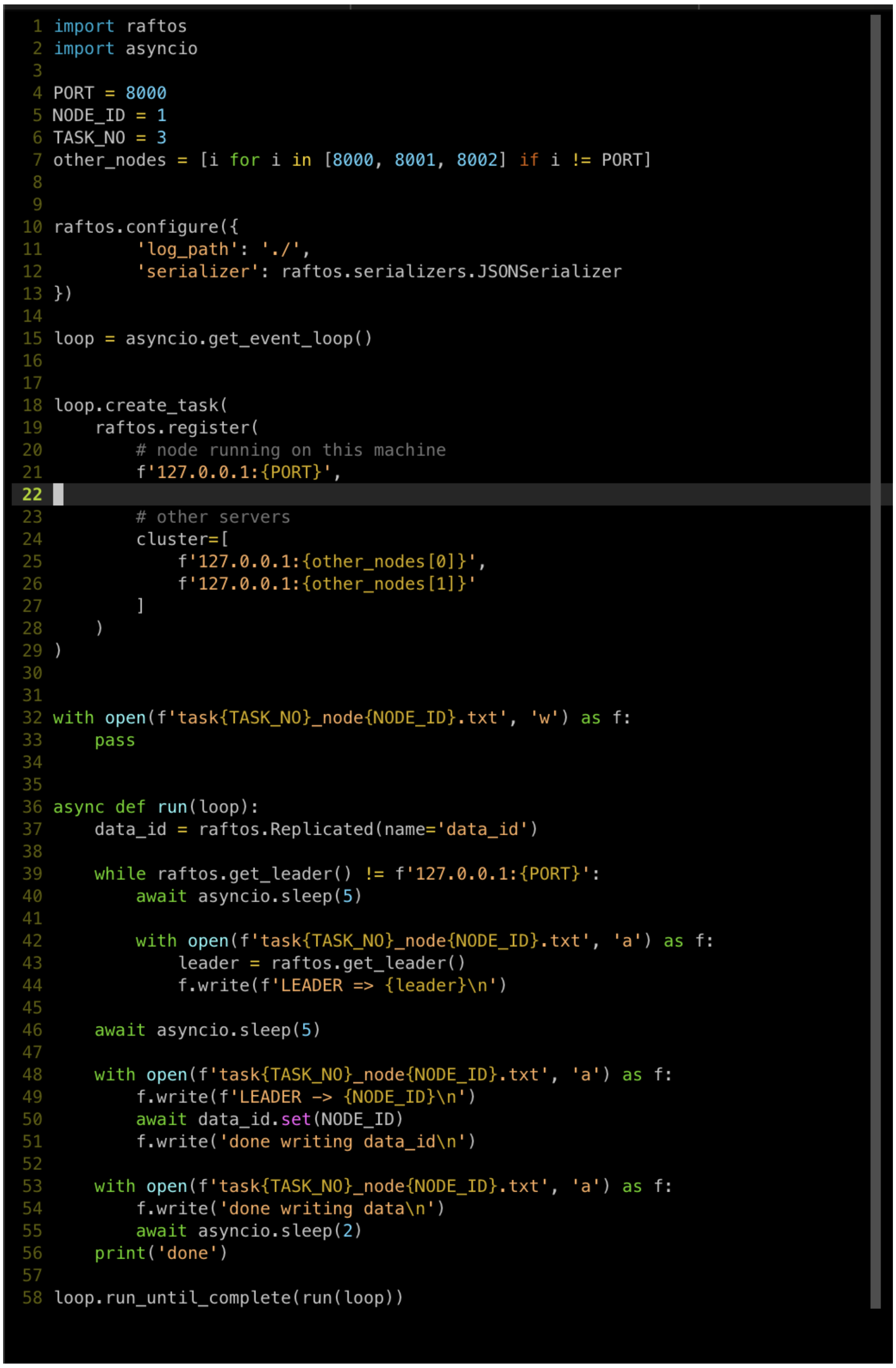


```
❯ cat *.txt

File: task2_node1.txt

LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => None
LEADER => 127.0.0.1:8000
LEADER IS NOW NODE 1

File: task2_node2.txt

LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => 127.0.0.1:8002
LEADER => None
LEADER => 127.0.0.1:8000
LEADER => 127.0.0.1:8000
LEADER => 127.0.0.1:8000
LEADER => None
LEADER => None
LEADER => None
LEADER => None

File: task2_node3.txt

LEADER => 127.0.0.1:8002
LEADER IS NOW NODE 3
```

Task 3: Verify that the data is logged and stored by all nodes involved

Step 1) Change the run function in all the three .py files to the one present in the below image. **Change TASK_NO is to 3 in all .py files**. Overall the .py files for task3 should like this:

Please write the f.write() statements as is, for evaluation purposes.



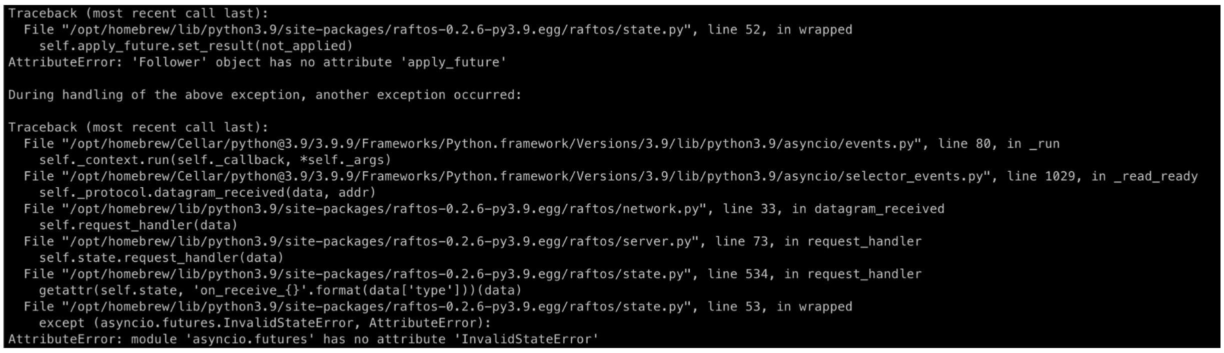
```python
import raftos
import asyncio

PORT = 8000
NODE_ID = 1
TASK_NO = 3
other_nodes = [i for i in [8000, 8001, 8002] if i != PORT]


raftos.configure({
        'log_path': './',
        'serializer': raftos.serializers.JSONSerializer
})

loop = asyncio.get_event_loop()


loop.create_task(
    raftos.register(
        # node running on this machine
        f'127.0.0.1:{PORT}',

        # other servers
        cluster=[
            f'127.0.0.1:{other_nodes[0]}',
            f'127.0.0.1:{other_nodes[1]}'
        ]
    )
)


with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'w') as f:
    pass


async def run(loop):
    data_id = raftos.Replicated(name='data_id')

    while raftos.get_leader() != f'127.0.0.1:{PORT}':
        await asyncio.sleep(5)

        with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'a') as f:
            leader = raftos.get_leader()
            f.write(f'LEADER => {leader}\n')

    await asyncio.sleep(5)

    with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'a') as f:
        f.write(f'LEADER -> {NODE_ID}\n')
        await data_id.set(NODE_ID)
        f.write('done writing data_id\n')

    with open(f'task{TASK_NO}_node{NODE_ID}.txt', 'a') as f:
        f.write('done writing data\n')
        await asyncio.sleep(2)
    print('done')

loop.run_until_complete(run(loop))
```

Step 2) Execute `rm 127*` to remove all .log, .storage and .state_machine files. The task may not run if they're present. Each time you rerun the experiment, you should remove the .log, .storage and .state_machine files.

Step 3) Run 1.py, 2.py and 3.py like you did in the previous task. Use the `tail -f <file>` command to check the logs.

Step 4) The leader should print "done" and complete executing.

If you get an error like this: `AttributeError: module 'asyncio.futures' has no attribute 'InvalidStateError'`, ignore it.



```
Traceback (most recent call last):
  File "/opt/homebrew/lib/python3.9/site-packages/raftos-0.2.6-py3.9.egg/raftos/state.py", line 52, in wrapped
    self.apply_future.set_result(not_applied)
AttributeError: 'Follower' object has no attribute 'apply_future'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "/opt/homebrew/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/python3.9/asyncio/events.py", line 80, in _run
    self._context.run(self._callback, *self._args)
  File "/opt/homebrew/Cellar/python@3.9/3.9.9/Frameworks/Python.framework/Versions/3.9/lib/python3.9/asyncio/selector_events.py", line 1029, in _read_ready
    self._protocol.datagram_received(data, addr)
  File "/opt/homebrew/lib/python3.9/site-packages/raftos-0.2.6-py3.9.egg/raftos/network.py", line 33, in datagram_received
    self.request_handler(data)
  File "/opt/homebrew/lib/python3.9/site-packages/raftos-0.2.6-py3.9.egg/raftos/server.py", line 73, in request_handler
    self.state.request_handler(data)
  File "/opt/homebrew/lib/python3.9/site-packages/raftos-0.2.6-py3.9.egg/raftos/state.py", line 534, in request_handler
    getattr(self.state, 'on_receive_{}'.format(data['type']))(data)
  File "/opt/homebrew/lib/python3.9/site-packages/raftos-0.2.6-py3.9.egg/raftos/state.py", line 53, in wrapped
    except (asyncio.futures.InvalidStateError, AttributeError):
AttributeError: module 'asyncio.futures' has no attribute 'InvalidStateError'
```

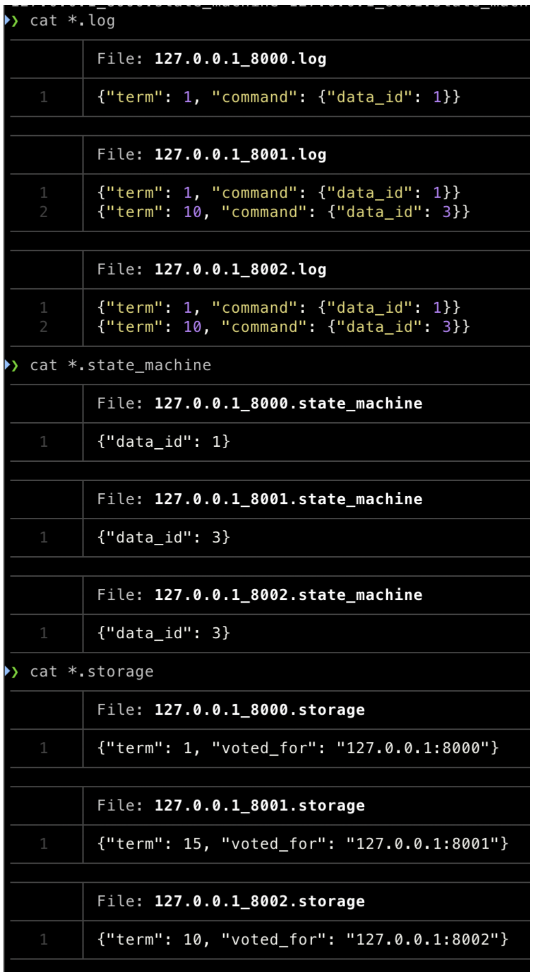
These errors may happen because Raftos is 5 years old and things have changed in python's asyncio since then.

Step 5) Observe that the node that becomes the new leader, also prints "done writing data" and finishes.

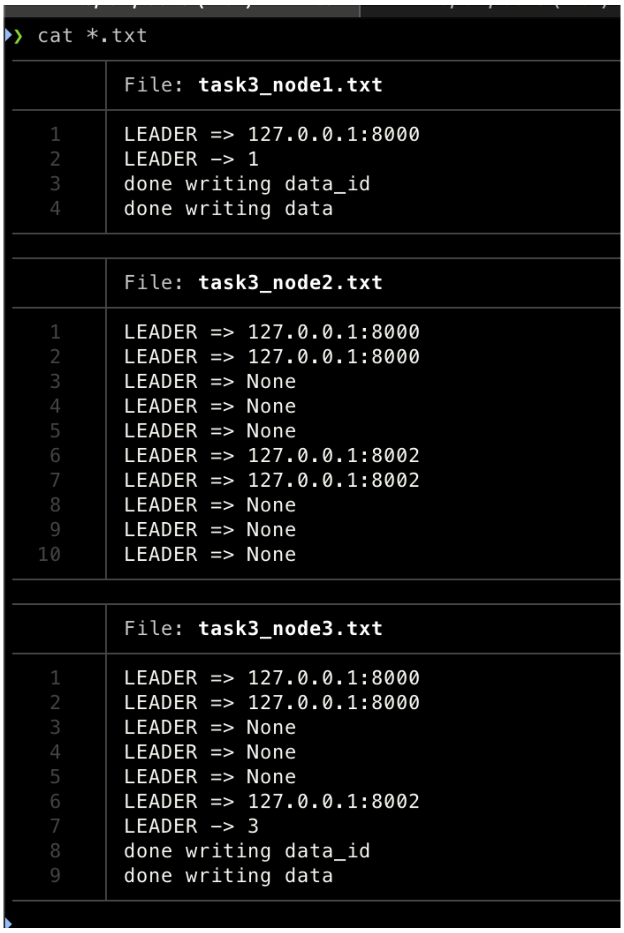
Step 6) Stop the final node using "Ctrl+C".

Step 7) `cat` all the .log, .state_machine, and .storage files and take a screenshot[3_a.jpg]. `cat` all the the .txt files as well and take a screenshot[3_b.jpg]

Example screenshots for this task are given below.



```
❯ cat *.log
      File: 127.0.0.1_8000.log
      {"term": 1, "command": {"data_id": 1}}

      File: 127.0.0.1_8001.log
      {"term": 1, "command": {"data_id": 1}}
      {"term": 10, "command": {"data_id": 3}}

      File: 127.0.0.1_8002.log
      {"term": 1, "command": {"data_id": 1}}
      {"term": 10, "command": {"data_id": 3}}
❯ cat *.state_machine
      File: 127.0.0.1_8000.state_machine
      {"data_id": 1}

      File: 127.0.0.1_8001.state_machine
      {"data_id": 3}

      File: 127.0.0.1_8002.state_machine
      {"data_id": 3}
❯ cat *.storage
      File: 127.0.0.1_8000.storage
      {"term": 1, "voted_for": "127.0.0.1:8000"}

      File: 127.0.0.1_8001.storage
      {"term": 15, "voted_for": "127.0.0.1:8001"}

      File: 127.0.0.1_8002.storage
      {"term": 10, "voted_for": "127.0.0.1:8002"}
```

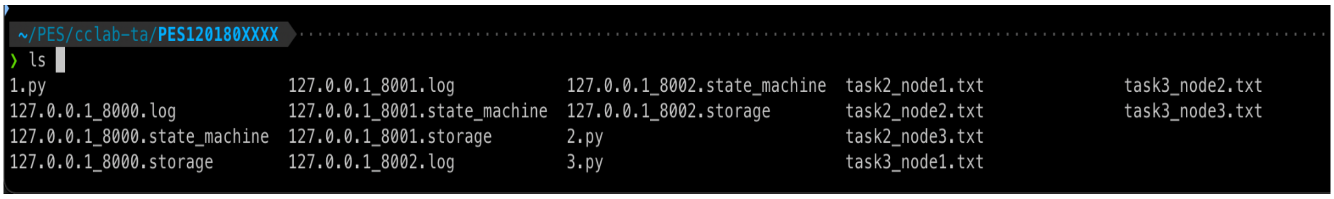


```
❯ cat *.txt
      File: task3_node1.txt
      LEADER => 127.0.0.1:8000
      LEADER -> 1
      done writing data_id
      done writing data

      File: task3_node2.txt
      LEADER => 127.0.0.1:8000
      LEADER => 127.0.0.1:8000
      LEADER => None
      LEADER => None
      LEADER => None
      LEADER => 127.0.0.1:8002
      LEADER => 127.0.0.1:8002
      LEADER => None
      LEADER => None
      LEADER => None

      File: task3_node3.txt
      LEADER => 127.0.0.1:8000
      LEADER => 127.0.0.1:8000
      LEADER => None
      LEADER => None
      LEADER => None
      LEADER => 127.0.0.1:8002
      LEADER -> 3
      done writing data_id
      done writing data
```

For evaluation:

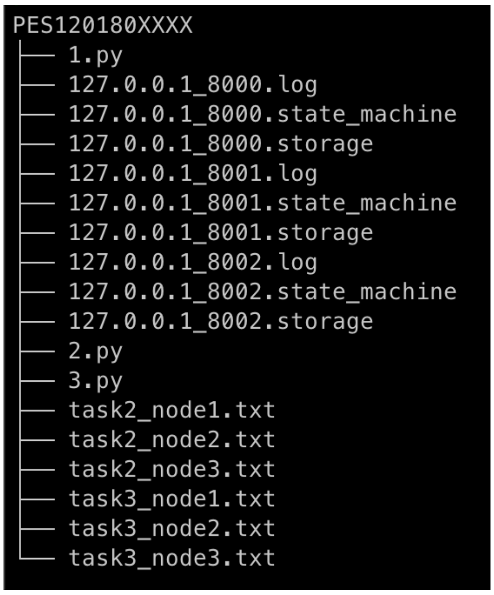
The submission has 2 parts:

1. Word doc - add all screenshots, file name:< your-srn>.docx
2. Zip file(<SRN>.zip - The zip file should consist of all code written and produced during this lab. That is, the .py files, the .txt files, the .storage files, the .state_machine files, the .log files should all be zipped together and submitted. Example of the directory that should be zipped:



```
~/PES/cclab-ta/PES120180XXXX
❯ ls
1.py                      127.0.0.1_8001.log             127.0.0.1_8002.state_machine  task2_node1.txt          task3_node2.txt
127.0.0.1_8000.log        127.0.0.1_8001.state_machine   127.0.0.1_8002.storage        task2_node2.txt          task3_node3.txt
127.0.0.1_8000.state_machine  127.0.0.1_8001.storage     2.py                          task2_node3.txt
127.0.0.1_8000.storage    127.0.0.1_8002.log             3.py                          task3_node1.txt
```

The PES120180XXXX directory should be zipped and submitted. Below image is a tree view of the directory.



```
PES120180XXXX
├── 1.py
├── 127.0.0.1_8000.log
├── 127.0.0.1_8000.state_machine
├── 127.0.0.1_8000.storage
├── 127.0.0.1_8001.log
├── 127.0.0.1_8001.state_machine
├── 127.0.0.1_8001.storage
├── 127.0.0.1_8002.log
├── 127.0.0.1_8002.state_machine
├── 127.0.0.1_8002.storage
├── 2.py
├── 3.py
├── task2_node1.txt
├── task2_node2.txt
├── task2_node3.txt
├── task3_node1.txt
├── task3_node2.txt
└── task3_node3.txt
```

References:

[1]: https://en.wikipedia.org/wiki/Consensus_(computer_science)

[2]: https://raft.github.io/

[3]: https://github.com/zhebrak/raftos