MPCS 52040 Distributed Systems
Project: Raft REST Message Queue (RRMQ)
Due: 3/3 @ 11:59 PM

The aim of this project is to create a distributed message queue using Raft. The system will expose a REST API for accessing the queue and maintain consistency between state by applying the Raft consensus algorithm.

The Raft REST Message Queue (RRMQ) will allow users to add (PUT) and consume (GET) messages from a specific topic.

- PUT(topic, message) the message will be appended to the end of the topic's message queue
- GET(topic) will pop the first message from the topic's queue and return the message

The message queue should follow a first-in-first-out (FIFO) mode.

Messages will be consumed (GET) only once by only one consumer. This implies that the nodes have to work together to ensure that the queue is kept consistent when messages are added (PUT) **and** when messages are consumed (GET).

RRMQ exemplifies many of the topics covered in this course, including distribution, consistency, replication, fault tolerance, consensus, and message brokers. As we have discussed in lectures, one of the challenges when developing a distributed message queue is the need to be available (that the service will stay online in the presence of failures) and consistent (that the service provides the same state).

In this project you will implement a distributed Message Queue using a REST API for client communications. As part of this project you will need to ensure that the RRMQ is distributed across several nodes and is fault tolerant.

1 Requirements

- The project should be implemented in Python 3.
- The project may be conducted by teams of up to two students. You are welcome to do the project alone if you like.
- You cannot use existing consensus libraries (e.g., Zookeeper or an implementation of Raft). You also should not build upon an existing message queue (e.g., RabbitMQ). If you are in doubt about using a particular library, please ask on Slack.
- Your RRMQ should be fault tolerant. That is, it should be resistant to failure of one or more nodes (up to $N/2–1$).
  - Raft Leader election must be tolerant of nodes crashing and restarting or network partitions.
  - Message production and consumption does not need to be tolerant of network partitions. That is, you can assume fail-stop failures (nodes may fail but they will not rejoin the system).
- RRMQ will implement a JSON-based interface as specified below.

2 Approach

The project includes three major components:

1. MESSAGE QUEUE: create a working single node message-queue implementation with the specified API
2. ELECTION: develop the raft election algorithm for multiple nodes
3. FAULT TOLERANCE: implement fault tolerance using the Raft consensus algorithm.

The three parts can be implemented in order without requiring any of the later functionality. We strongly suggest you implement these components in order and thoroughly test each step before moving to the next.

3 Submission

The entire project is due on 3/3 @ 11:59 PM. You should submit via the GitHub classroom link: `https://classroom.github.com/a/6KKgvX0T`

Please include the following required files:

- a src folder with node.py and any other required files. we will use node.py to start your node as specified.
- technical_report.md, testing_report.md, and contributions_report.md as specified in the Report section.
- requirements.txt listing all the modules required by your service, remember to ask on Slack if you are unsure about using a particular module.
- a test folder with your tests in submission_test.py and any other required files. these tests should be based on the provided test framework.

You may add other folders or files.

Please structure your repository as follows:

```
src/
  any_other_file.py
  node.py
test/
  any_other_file_test.py
  submission_test.py
technical_report.md
testing_report.md
contributions_report.md
requirements.txt
```

3.1 Report

You should submit a technical_report.md (ideally less than 16kB) that outlines:

- For each of the 3 parts:
  - how you implemented that part and what design decisions you made
  - possible shortcomings of your implementation
- All sources used in developing your implementation

You should submit a testing_report.md (ideally less than 8kB) that outlines:

- For each of the 3 parts:

  - how you tested that part specifically
  - any possible shortcomings of your testing approach

You should submit an contributions_report.md (less than 1kB) with:

- Names and e-mail addresses of all the students in the group
- A description of which parts each student worked

3.2 Grading

Grading of this project is out of 100 points:

- 10 points: single node put/get implementation for messages and topics
- 30 points: raft election and the relative status API
- 10 points: raft election with fault recovery
- 30 points: log replication between nodes and fail-stop tolerance
- 15 points: technical_report.md
- 05 points: testing_report.md and tests

3.3 Integration tests for grading

We will grade the project using our integration testing framework. This will allow us to automatically stop and start nodes as well as PUT and GET messages from topics.

We will provide the testing framework and a set of test cases so that you can ensure your implementation follows the defined interfaces and works with the testing framework. It will also serve as a framework for you to write your own tests.

We will release a set of test cases for each part of the project.

4 Example approach

When developing your implementation we suggest you review the Raft website and the original Raft paper. The paper is written with the goal of being understandable and it provides the most clear and accessible documentation of the algorithm.

Below we outline one approach to this project.

4.1 Create a single node Message Queue

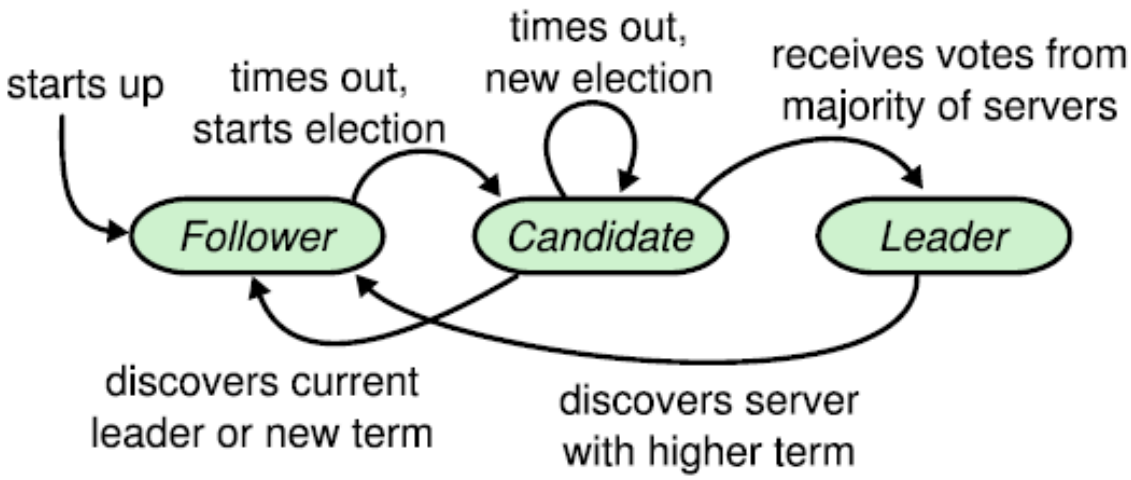
First you should create single REST service that implements the PUT and GET operations described above. You will need to define a data structure for storing the message queue and the topics (e.g., a Python dictionary and list) and the functions to add/retrieve messages to/from that structure. You will need to implement functionality to parse the JSON requests and return JSON responses. You should create a client to test different scenarios (e.g., creating topics, putting messages, getting messages that don't exist, etc.) so that in subsequent steps of the project you can identify consistency issues.

4.2 Implement the leader election algorithm

In the second step you should extend your service so that several instances can be deployed (you can deploy instances on different ports on your computer). To do so, you will implement the Raft algorithm First you will need to determine one node to be the "leader" and ensure that your client always coordinates with the leader. Note: followers can simply reply to clients saying they are not the leader. You will need to provide

each instance of the service with a configuration file (described below) that specifies the other nodes in the system.

The best source of information for implementing Raft's leader election algorithm is in the Raft paper. It is important to note that this is a fairly precise specification of the algorithm and thus requires that you follow it accurately. The basic lifecycle of the algorithm is depicted in Figure 4.2. You will need to implement these transitions between the various states (follower, candidate, leader)



You will need to add new message types (beyond the aforementioned 'PUT' and 'GET' messages) to your server such that it can exchange information with other nodes in your system. Information on the address of the other nodes can be passed to each node at startup through a configuration file. In the Raft paper it refers to RPC methods. You do not need to implement these methods using RPC, you can use any of the messaging technologies used in class with your own messaging format (ZMQ, RPC, REST, and others). Briefly, you will need to do the following things:

1. Implement the Raft election protocol. You will need to implement much of the state described in Figure 2 of the Raft paper, store the state of the server (candidate, follower, leader), and implement the transitions between roles.
2. Extend your MQ interface such that only the leader will reply to clients PUT/GET requests of messages/topics. If a node is not the leader it will reply with a failure. The Status endpoint should still work for any node.
3. Add the timeout capability to your server such that it can detect leader failure. That is, if a node has not heard from the leader in some period of time, it will start a new election process.
4. Add a heartbeat message (Using the AppendEntries message) such that a leader can periodically notify all other nodes that it is still alive.
5. Test your implementation by removing leaders and validating that a new leader takes over.

Note: many of these actions are asynchronous and thus require some way of enabling concurrency. You should use threads for this purpose.

Note: the leader election timeout stated in the Raft paper is very short. For testing, you will want to configure the timeout range such that you can more easily track the flow of messages in your system. For the final version you should decrease the timeout to something near the stated time.

4.3 Implement the distributed replication model

Now you have a leader that is capable of servicing all incoming requests you will need to implement the distributed replication model to enable the leader to distribute updates to other nodes in the system. The basic model, as outlined in the Raft paper, relies on the leader recording each command to a local log (i.e., a list) and then sending messages to other nodes in the system to distribute that log. Note, when distributing the log you will need to share it to a quorum of other nodes so as to ensure that there is no potential for inconsistency. You will need to do the following steps:

1. Implement the state machine for each node.
2. Implement the transaction log and make sure that the leader correctly stores operations in this log.
3. Enhance the AppendEntries message so that it is able to distribute log data to other servers.
4. Implement functionality to make sure that a quorum is in agreement before committing the log changes to the local state machine.

4.4 Testing

Raft is an intricate algorithm and there are many failure conditions. You will need to create both a set of automated tests and conduct a range of manual tests to ensure that your implementation works correctly.

We have provided a set of initial tests to ensure that you follow the testing framework and to help you validate that your interfaces match those described.

`https://github.com/mpcs-52040/2024-RRMQ-Tests`

5 Implementation and API Specifications

The following section outlines the interfaces you should follow when implementing your system.

5.1 Starting the server

Start the server with the following command:

```
> python3 src/node.py path_to_config index
```

Where the input arguments are as follows.

- path_to_config is the path to a JSON file with a list of the ip and port of all the nodes in the RRMQ (relative to the root folder of the repository).
- index is the index of the current server in that JSON list of addresses that will be used to let the server know its own IP and port.

For example, a system with 3 local nodes on the ports 8567, 9123, 8889 would have the following config.json file

```
{"addresses":
  [
    {"ip": "http://127.0.0.1", "port": 8567},
    {"ip": "http://127.0.0.1", "port": 9123},
    {"ip": "http://127.0.0.1", "port": 8889}
  ]
}
```

And we would start this system with the following commands

```
> python3 src/node.py config.json 0     # will be on http://127.0.0.1:8567
> python3 src/node.py config.json 1     # will be on http://127.0.0.1:9123
```

```
> python3 src/node.py config.json 2     # will be on http://127.0.0.1:8889
```

5.2 REST API

The REST API has three endpoints: Topic, Message, and Status. Note: in addition to the response types described here, your REST API should return appropriate HTTP codes in response to different events.

5.2.1 Topic

The topic endpoint is used to create a topic and retrieve a list of topics. Topics are simply string names.

PUT /topic

Used to create a new topic.

Flask endpoint:

```
@app.route('/topic', methods=['PUT'])
```

Body:

```
{'topic' : str}
```

Returns:

```
{'success' : bool}
```

Returns True if the topic was created, False if the topic could not be created (e.g., if the topic name already exists).

GET /topic

Used to get a list of topics

Flask endpoint:

```
@app.route('/topic', methods=['GET'])
```

Returns:

```
{'success' : bool, 'topics' : [str]}
```

If there are no topics it returns an empty list.

5.2.2 Message

The message endpoint is used to add a message to a topic and get a message from a topic.

PUT /message

Used to add a message to a topic.

Flask endpoint:

```
@app.route('/message', methods=['PUT'])
```

Body:

```
{'topic' : str, 'message' : str}
```

Returns:

```
{'success' : bool}
```

returns True if the message was added and False if the message could not be added (e.g., if the topic does not exist).

GET /message

Used to pop a message from the topic. Notice that the topic name is encoded in the URL.

Flask endpoint:

```
@app.route('/message/<topic>', methods=['GET'])
```

Returns:

```
{'success' : bool, 'message' : str}
```

Returns False if the topic does not exist or there are no messages in the topic that haven't been already consumed

5.2.3 Status

The status endpoint is used for testing the leader election algorithm.

GET /status

Flask endpoint:

```
@app.route('/status', methods=['GET'])
```

Returns:

```
{'role' : str, 'term' : int}
```

Role may be one of three options: Leader, Candidate, Follower

Term is the node's current term as an integer.