CS4103 P2 - Leader elections and mutual exclusion

160004864

5 May 2020

1 Introduction

For this practical we were required to design and implement a distributed system for an integrated library system that makes use of coordination and resource access algorithms. Coordination should be achieved by using the **Bully Algorithm** for leader election and resource access should be controlled by using a **centralised algorithm** for mutual exclusion. Furthermore, the distributed system must provide clients with the ability to make updates in a library record, specifically supporting borrow and return book operations in a transparent manner.

2 Functionality Implemented

The following is a list of the functionality implemented as part of this assignment with a brief description:

- **Communication**: A system with 5 multi-threaded nodes was created using the Python programming language, in which communication is achieved using available Socket API only.
- **Ping Node**: The nodes provide a text-based admin interface which allows an admin to ping another node.
- **Logs**: Each node maintains a log recording its activity such as the message it receives and sends.
- **Crash simulation**: The admin interface offers the admin the option to simulate a crash which terminates the node process.
- **Leader election**: A node may be instructed by its administrator to initiate a coordinator election which is done using the Bully Algorithm for leader election. Extension - Furthermore, the election process can be triggered if a node detects that a node is down, for example when the ping request times out. In addition the system allows for multiple nodes to start an election and it is automatic, it does not require the need for an admin user input.
- **Receive/Send Posts**: The system allows clients to make update requests for operations on the library records. Updates can either be borrowing or returning a book. Access to the resource is controlled by a centralised mutual exclusion algorithm.

With reference to the specification, all the functionality required for Parts 1, 2 and 3 was implemented.

3 Design

In this section I will be giving a description and justification for the implemented functionality.

3.1 Node Setup

Nodes are the the server nodes that make up our distributed system and they are implemented in the `node.py` module. Each of the 5 nodes have a unique ID which is used to identify them and they are assigned a unique hostname-port pair (as found in the nodes.csv file) that is used to set up the server socket on which they will be listening for requests on, whether these are request from other nodes for intra-system communication and coordination or these are requests coming from clients. For every new connection, nodes will spawn a new thread which performs the `handle_client` method which processes the message received and depending on its type (see 3.3) the server will carry out the appropriate functionality. Nodes can act either as servers or clients depending on the kind of message they will be sending. Simply, if they are initiating a request they reach out to the other nodes as clients while their server sockets listens for such requests either from other nodes or clients external to the system.

In addition, nodes provide a textual user interface to their administrator that is handled by a different thread which performs the `admin_interface.py` method. This interface allows the admin to carry out the following operations:

- Ping a node: The admin is prompted for the ID of the node they wish to ping and the ping request is sent to that node.
- Initiate election: The admin can initiate a leader/coordinator election.
- Disconnect node: The admin can disconnect a node from the system and shut it down. This simulates a crash.

3.2 Client Setup

The implementation of the client functionality can be found ine the `client.py` module. Contrary to nodes, clients that will be making book library updates to the distributed system only maintain a client socket as they will not be listening for any requests coming from anywhere. Clients are identified with a unique username, making them recognisable customers in the library records. The `client_interface` method provides a text-based interface for the client, allowing them to choose from the following operations:

- Borrow a book: This prompts the client to enter the book id of the book they wish to borrow. Following that the client connects to a node in the system to which it sends an update request a borrow operation.
- Return a book: This prompts the client to enter the book id of the book they wish to return. Following that, the client connects to a node in the system to which it sends an update request for a return operation.
- Disconnect: This shuts down the client program.

3.3 Message Protocol

In order to facilitate smooth communication, I have designed a message protocol which is followed by the nodes and the clients that are part of our distributed system. Each message is composed of three parts:

- Sender ID: This is the ID of the sender.
- Message Type: This is the type of the message. More described below.
- Message Body: This is the main body of the message, used if additional information needs to be communicated.

Upon receiving a message nodes and clients can carry out the appropriate functionality depending on what the type of the message was. The message types used as part of the protocol are defined in the `node_messages.py` file. The following are the types of messages that can be send between nodes:

- `PING_MESSAGE`: Sent by the node that initiates a ping request is initiated. Received by the node that is on the receiving end of the ping.
- `PING_RESPONSE`: Sent as a response to a `PING_MESSAGE`.
- `ELECTION_MESSAGE`: Sent by the node initiating a leader election. Received by all nodes with a higher ID than the ID of the initiator.
- `OK_MESSAGE`: Sent as a response to an `ELECTION_MESSAGE`.
- `COORDINATOR_MESSAGE`: Sent by the node that won the leader/coordinator election. Received by all the other nodes.
- `REQUEST_MESSAGE`: Sent by the node that is requesting access to the book library resources. Received by the coordinator.
- `REQUEST_OK`: Sent by the coordinator to the node which has been granted access to the book library resources.
- `RELEASE_MESSAGE`: Sent by the node which has finish accessing the book library resources. Received by the coordinator.
- `DISCONNECT_MESSAGE`: This message is not really ever sent over the network. It is a mock message which is 'sent' from the admin to their node, to disconnect the node and shut it down.

The messages exchanged between nodes and clients are the following:

- `BORROW_UPDATE`: Sent by the client that wishes to perform a borrow operation. Received by the node the client is connected with.
- `BORROW_SUCCESS`: Sent as a response to a `BORROW_UPDATE` message, if the borrow operation is successful.
- `BORROW_FAIL`: Sent as a response to a `BORROW_UPDATE` message, if the borrow operation is unsuccessful.
- `RETURN UPDATE`: Sent by the client that wishes to perform a return operation. Received by the node the client is connected with.
- `RETURN_SUCCESS`: Sent as a response to a `RETURN_UPDATE` message, if the return operation was successful.
- `RETURN_FAIL`: Send as a response to a `RETURN_UPDATE` message, if the return operation was unsuccessful.

3.4 Ping

A ping request is initiated by a node's administrator. The node will act as a client and it will connect to the node that the administrator chose to ping, and send a `PING_MESSAGE` to it, a functionality that is carried out in the `ping` method. The pinged node will receive this message through its server socket and it will respond with a `PING_RESPONSE` message. If this is received, the client node will print the response and carry on. If the response times out however, it will interpret the node as being down and thus it will initiate coordinator elections.

3.5 Elections

Elections are carried out using the Bully Algorithm, as described here[1] and implemented in the `initiate_election` method. Elections can be initiated in several ways. The admin of a node may instruct it to initialise the election or it may be the result of a ping request timing out. Elections are initiated by a node, now acting as a client, by sending an `ELECTION_MESSAGE` to the nodes that have a higher ID than itself. The node will then wait for an `OK_MESSAGE` from any of these nodes. If this times out then it will be the winner of the election and it will send a `COORDINATOR_MESSAGE` to all of the nodes, announcing that it is the new coordinator. If it receives an `OK_MESSAGE`, it will wait for a `COORDINATOR_MESSAGE`. If it receives this, it sets the new coordinator as the message's sender and carries on. If this times out however, it will initiate another election.

An important part of this process is that nodes that have higher id's conduct their own elections in the same way. For this reason when a node receives an `ELECTION_MESSAGE` through its server socket, it will call `initiate_election` as well and it will carry out the same process as it was described above.

To better understand this process, I envisioned the stages a node can find itself during an election as a set of states and the messages as transitions between said states, as depicted in the following visualisation:

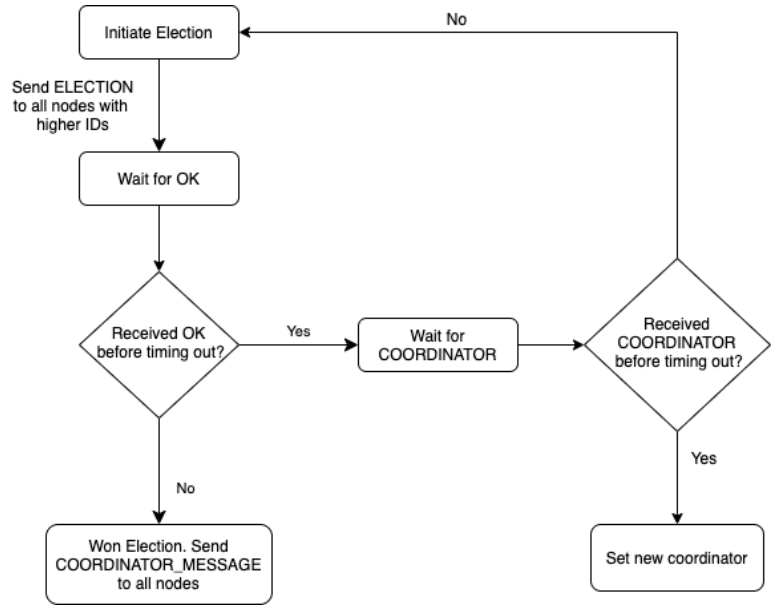


Figure 1: Election states and messages, visualised.

3.6 Library Record

The library record consists of a list of instances of the `Book` class, defined in the `book.py` module. Its class diagram is as follows:

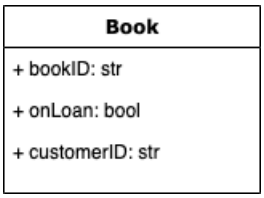


Figure 2: Book class diagram.

Where:

- `bookID`: is the unique id for the book. With the current settings, this will be between 1 and 10 (inclusive).
- `onLoan`: holds whether the book is on loan or on shelf.
- `customerID`: keeps the username of the customer that has borrowed the book. If on shelf, this is an empty string.

The list is created by the `create_books.py` module and it is saved as a file within the `resources` directory. This module should only be called to initialise the list or to wipe out any existing records.

The `access_books.py` module provides a number methods for accessing modifying and saving the updated book list. They also perform the necessary validation checks, for example checking that a book to be borrowed is on shelf, and whether the book to be returned is on loan, and if it is, it was taken out by the customer making the request. These method are used by the node, once they are granted access to the resource.

3.7 Borrow & Return Updates

The client can make a borrow or a return update by sending either a `BORROW_UPDATE` or a `RETURN_REQUEST` message to a node in the distributed system. The node records this request, as a tuple of `(<operation_type>, <request_msg>), <connection>`) where `operation_type` can either be 'borrow' or 'return', `request_msg` is the message as received from the client and `connection` is the connection between the node and the client, which is kept open until the request is fulfilled. This tuple is then added to a FIFO queue that each node maintains named `CLIENT_OPERATION_QUEUE`, which holds the requests sent by clients. This was implemented in this way as it allows as node to carry on with other operations, and return to fulfilling these requests when the node has been notified by the coordinator that is been granted access to the library record.

Once access has been granted to this node (see 3.8), the node will perform the operation at the front of the `CLIENT_OPERATION_QUEUE`, which depending on the type of operation it is, it will either call the `perform_borrow` method or the `perform_return` method. These make use of the access methods provided by `access_books.py` to attempt the updates for borrowing or returning a book. If these succeed, the node will respond to the client with either a `BORROW_SUCCESS` or `RETURN_SUCCESS` message to notify the client that their request has been completed. If these fail the node will respond to the client with either a `BORROW_FAIL` or `RETURN_FAIL` message.

3.8 Resource Access

Before a node can access the book list resource to make the updates as requested by a client, they must be granted access by the coordinator node which uses a centralised algorithm for mutual exclusion. Upon receiving an update request from a client, the node connects to the server socket of the coordinator node and it sends a `REQUEST_MESSAGE`. When the coordinator node receives this message, it will check whether the lock on the resource is currently being held by another node. If it is not, it will call the `grant_access` method, in which it will then act as a client and send the requesting node a `REQUEST_OK` message. When this message is received by the requesting node it will call the `perform_client_operation` method, which will carry out the operation that is at the front of its `CLIENT_OPERATION_QUEUE`. In the case where a lock is currently held by another node, the request will be recorded stored in a FIFO queue kept by the coordinator named `REQUEST_QUEUE`. When the node holding the lock completes its operation on the resource, it will send a `RELEASE_MESSAGE` to the coordinator. This will signal the coordinator that this node has finished working with the resource and at this point it will pop the first element of `REQUEST_QUEUE`, to which it will send a `REQUEST_OK` and it will grant is access to the resource using `grant_access` method.

3.9 Logging

Logging is achieved using the classes defined in the `logger.py` module. The logging functionality for the nodes is handled by the `NodeLogger` class, which keeps a file for each node within the `logs` directory. The logs contain details of the most important activity for each node, such as the messages it sends and receives. I have decided to keep logs for the clients as well in a similar fashion, which are handled by instances of the `ClientLogger` class.

4 Examples and Testing

4.1 Ping

In this example we will include examples of the ping functionality working. In the following example the admin of node with `ID=1` instructs the node to ping the node with `ID=2`:

```
[2020-05-06 17:21:55 | 1 ] Listening on 10.5.17.178:5001
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
1
[ADMIN INTERFACE] Enter ID of node you would like to ping
2
[2020-05-06 17:22:07 | 1 ] Receive from Admin user: PING 2
[2020-05-06 17:22:07 | 1 ] Receive from 2: Hello from Node 2
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
```

(a) Node `ID=1`

```
[2020-05-06 17:21:59 | 2 ] Listening on 10.5.17.178:5002
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 17:22:07 | 2 ] 10.5.17.178:60512 connected.
[2020-05-06 17:22:07 | 2 ] Receive from 1: PING 2
[2020-05-06 17:22:07 | 2 ] Send to 1: PING_RESPONSE 2
```

(b) Node `ID=2`

Figure 3: Ping functionality example.

As we can see, the admin of node with `ID=1` chooses to ping node with `ID=2`. It then sends a `PING_MESSAGE` to node `ID=2` which responds with a `PING_RESPONSE` message, as required.

4.2 Crash Simulation

In this example we will be demonstrating the crash simulation feature, which is activated when admins instruct their node to shut down:

```
[2020-05-06 17:32:53 | 2 ] Listening on 10.5.17.178:5002
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
3
[2020-05-06 17:32:57 | 2 ] Receive from Admin user: !DISCONNECT
Killed: 9
```

Figure 4: Crash simulation functionality example.

As we can see the admin chooses the third option, 'Disconnect' which causes the node process to be killed, simulating a crash.

4.3 Leader Election

We have seen several ways in which an election may be initiated. In the first example we will look at an example where the admin of the node with `ID=3` initiates an election:

```
[2020-05-06 17:38:06 | 3 ] Listening on 10.5.17.178:5003
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
2
[2020-05-06 17:38:13 | 3 ] Send to 4: ELECTION
[2020-05-06 17:38:13 | 3 ] Send to 5: ELECTION
[2020-05-06 17:38:13 | 3 ] Receive from 4: OK 4
[2020-05-06 17:38:13 | 3 ] Receive from 5: OK 5
[2020-05-06 17:38:23 | 3 ] 10.5.17.178:60606 connected.
[2020-05-06 17:38:23 | 3 ] Receive from 5: COORDINATOR 5
[2020-05-06 17:38:23 | 3 ] 10.5.17.178:60607 connected.
[2020-05-06 17:38:23 | 3 ] Receive from 5: COORDINATOR 5
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
```

Figure 5: Admin initiating election example.

As we can see the admin has entered option 2 (Election) in the admin interface which caused an election to be initiated. The node sent an `ELECTION_MESSAGE` to nodes with `ID=4` and `ID=5` which responded with an `OK_MESSAGE`. Following that we see that the subsequent elections that were initiated by the nodes with higher IDs results resulted in the node with the highest id `ID=5` to win, as it sent a `COORDINATOR` message.

There is a reason why node with `ID=3` received the `COORDINATOR` message twice. This is because node with `ID=5` participated in 2 elections - one initiated by node with `ID=3` and one initiated by node with `ID=4` upon receiving the very first `ELECTION_MESSAGE` from node with `ID=3`. The following is the log produced for node with `ID=5` for the same example:

```
[2020-05-06 17:37:57 | 5 ] Listening on 10.5.17.178:5005
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 17:38:13 | 5 ] 10.5.17.178:60599 connected.
[2020-05-06 17:38:13 | 5 ] Receive from 3: ELECTION
[2020-05-06 17:38:13 | 5 ] Send to 3: OK
[2020-05-06 17:38:13 | 5 ] 10.5.17.178:60600 connected.
[2020-05-06 17:38:13 | 5 ] Receive from 4: ELECTION
[2020-05-06 17:38:13 | 5 ] Send to 4: OK
[2020-05-06 17:38:23 | 5 ] Send to 1: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 1: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 2: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 2: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 3: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 3: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 4: COORDINATOR 5
[2020-05-06 17:38:23 | 5 ] Send to 4: COORDINATOR 5
```

Figure 6: Node receiving multiple election requests.

We have seen that another way an election can be initiated is when a ping request times out or when it is down. The following is an example produced when a timeout was forced for a ping request:

```
[2020-05-06 17:55:32 | 3 ] Listening on 10.5.17.178:5003
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
1
[ADMIN INTERFACE] Enter ID of node you would like to ping
5
[2020-05-06 17:56:26 | 3 ] Receive from Admin user: PING 5
[ADMIN INTERFACE] Timeout: initiate election.
[2020-05-06 17:56:36 | 3 ] Send to 4: ELECTION
[2020-05-06 17:56:36 | 3 ] Send to 5: ELECTION
[2020-05-06 17:56:36 | 3 ] Receive from 4: OK 4
[2020-05-06 17:56:46 | 3 ] 10.5.17.178:60732 connected.
[2020-05-06 17:56:46 | 3 ] Receive from 5: COORDINATOR 5
[2020-05-06 17:56:46 | 3 ] 10.5.17.178:60735 connected.
[2020-05-06 17:56:46 | 3 ] Receive from 5: COORDINATOR 5
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
```

Figure 7: Election initiated after ping request timing out.

The ping request to node with `ID=5` timed out and so elections were initiated. However, none of the election messages were forced to time out so elections carried on as normal.

The following is an example of an election where the current coordinator, node `ID=5`, shuts down and an election is then initiated:

```
[2020-05-06 18:04:04 | 5 ] Listening on 10.5.17.178:5005
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
3
[2020-05-06 18:04:23 | 5 ] Receive from Admin user: !DISCONNECT
Killed: 9
```

(a) Node `ID=5`

```
[2020-05-06 18:04:15 | 3 ] Listening on 10.5.17.178:5003
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
1
[ADMIN INTERFACE] Enter ID of node you would like to ping
5
[ADMIN INTERFACE] Exception: initiate election.
[2020-05-06 18:04:27 | 3 ] Send to 4: ELECTION
[2020-05-06 18:04:27 | 3 ] Could not connect to 5
[2020-05-06 18:04:27 | 3 ] Receive from 4: OK 4
[2020-05-06 18:04:37 | 3 ] 10.5.17.178:60793 connected.
[2020-05-06 18:04:37 | 3 ] Receive from 4: COORDINATOR 4
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
```

(b) Node `ID=3`

Figure 8: Election initiated after ping request detected a node was offline.

Node `ID=3` detected that node `ID=5` was offline, and it thus intited elections. The node with the highest ID was now node `ID=4` which became the new coordinator.

In the following example I will be testing the system when a time out is forced on the `OK_MESSAGE`.

```
[2020-05-06 18:11:25 | 3 ] Listening on 10.5.17.178:5003
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
2
[2020-05-06 18:12:32 | 3 ] Send to 4: ELECTION
[2020-05-06 18:12:32 | 3 ] Send to 5: ELECTION
[2020-05-06 18:12:42 | 3 ] Send to 1: COORDINATOR 3
[2020-05-06 18:12:42 | 3 ] Send to 2: COORDINATOR 3
[2020-05-06 18:12:42 | 3 ] Send to 4: COORDINATOR 3
[2020-05-06 18:12:42 | 3 ] Send to 5: COORDINATOR 3
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 18:13:22 | 3 ] 10.5.17.178:60855 connected.
[2020-05-06 18:13:22 | 3 ] Receive from 4: COORDINATOR 4
[2020-05-06 18:13:22 | 3 ] 10.5.17.178:60856 connected.
[2020-05-06 18:13:22 | 3 ] Receive from 5: COORDINATOR 5
[2020-05-06 18:14:02 | 3 ] 10.5.17.178:60864 connected.
[2020-05-06 18:14:02 | 3 ] Receive from 5: COORDINATOR 5
```

(a) Node `ID=3`

Figure 9: Forcing `OK_MESSAGE` to time out.

As we can see from the above, the `OK_MESSAGE` times out and so node with `ID=3` declares itself the winner of the election. However, when the other nodes eventually wake up, they 'bully' the other nodes into submission, with node `ID=5` eventually being the final coordinator.

The last example for our election functionality will be the example produced when the `COORDINATOR_MESSAGE` times out:

```
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
2
[2020-05-06 18:20:09 | 3 ] Send to 4: ELECTION
[2020-05-06 18:20:09 | 3 ] Send to 5: ELECTION
[2020-05-06 18:20:09 | 3 ] Receive from 5: OK 5
[2020-05-06 18:20:19 | 3 ] Send to 4: ELECTION
[2020-05-06 18:20:19 | 3 ] Send to 5: ELECTION
[2020-05-06 18:20:19 | 3 ] Receive from 5: OK 5
[2020-05-06 18:20:29 | 3 ] Send to 4: ELECTION
[2020-05-06 18:20:29 | 3 ] Send to 5: ELECTION
```

(a) Node `ID=3`

Figure 10: Forcing `COORDINATOR_MESSAGE` to time out.

In the above, we can see that while an `OK_MESSAGE` is received from node `ID=5`, a `COORDINATOR_MESSAGE` is never received and for that reason it times out and an election is re-initiated, as required. If the `OK_MESSAGE` keeps being received while the `COORDINATOR_MESSAGE` always times out, the elections will be re-initiated indefinitely.

4.4 Receive/Send Posts

Within this section we will be going over examples of all the functionality related to accessing the library records. We will be starting with a 'clean' library record, that is all the books are on shelf.

In the following example, a client connects to node with `ID=1` and it sends it a `BORROW_UPDATE` message for borrowing book with `ID=1`.

```
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
1
[CLIENT INTERFACE] Enter ID of the book you would like to borrow.
1
[2020-05-06 18:48:39 | Alpha ] Send to 1: BORROW_UPDATE
[2020-05-06 18:48:39 | Alpha ] Receive from 1: Borrow Successful
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
```

(a) Client `ID=Alpha`

```
[2020-05-06 18:48:34 | 1 ] Listening on 10.5.17.178:5001
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 18:48:39 | 1 ] 10.5.17.178:61173 connected.
[2020-05-06 18:48:39 | 1 ] Receive from Alpha: BORROW_UPDATE 1
[2020-05-06 18:48:39 | 1 ] Send to 5: REQUEST
[2020-05-06 18:48:39 | 1 ] 10.5.17.178:61175 connected.
[2020-05-06 18:48:39 | 1 ] Receive from 5: REQUEST OK 1
[RESOURCE] Lending book 1 to Alpha
[2020-05-06 18:48:39 | 1 ] Update Q - Book 1: On loan.
[2020-05-06 18:48:39 | 1 ] Send to 5: RELEASE
```

(b) Node `ID=1`

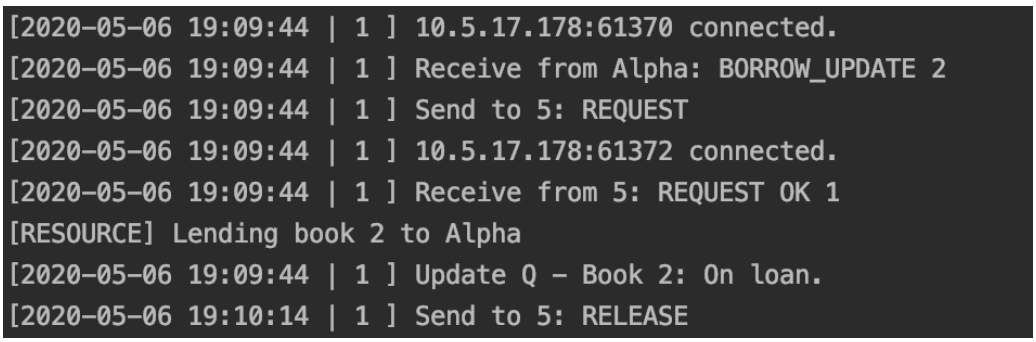
```
[2020-05-06 18:48:31 | 5 ] Listening on 10.5.17.178:5005
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 18:48:39 | 5 ] 10.5.17.178:61174 connected.
[2020-05-06 18:48:39 | 5 ] Receive from 1: REQUEST Q
[2020-05-06 18:48:39 | 5 ] Send to 1: REQUEST OK
[2020-05-06 18:48:39 | 5 ] 10.5.17.178:61176 connected.
[2020-05-06 18:48:39 | 5 ] Receive from 1: RELEASE Q
```

(c) Node `ID=5`

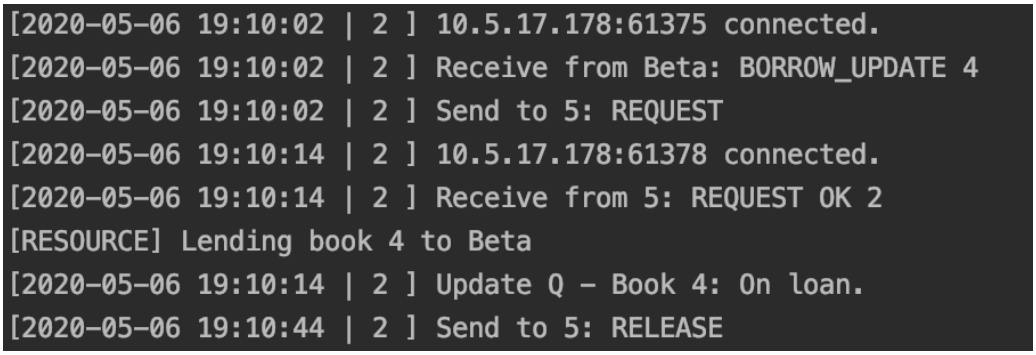
Figure 11: Resource access example.

As we can see the client is prompted to enter an option and they choose to borrow the book with `ID=1`. As a result it sends a `BORROW_UPDATE` message to node with `ID=1`, which eventually responds with a `BORROW_SUCCESS` message. Between these two events however, we can see that node with `ID=1` sends a `REQUEST_MESSAGE` to the coordinator, node `ID=5`. The coordinator then sends a `REQUEST_OK` message back to node `ID=1` which prompts it to access the resource and perform the required updates. Furhtermore, upon finishing the operation, node `ID=1` sends a `RELEASE_MESSAGE` to the coordinator indicating that it is done working accessing the resource.

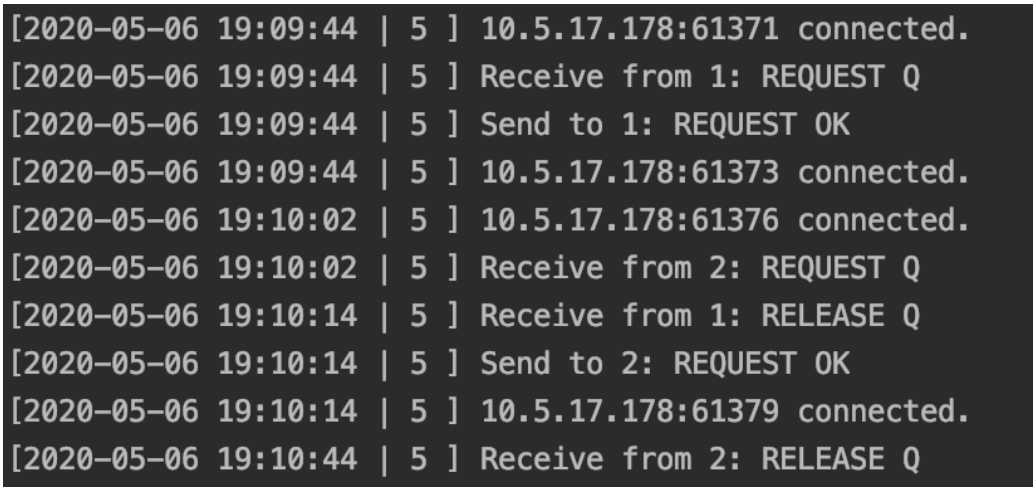
In this next example, we will be adding some delay between when the node completes the resource access to the time it sends a `RELEASE_MESSAGE` to the coordinator, to show how the queue works. We will have two clients connected to the system, client `Alpha` connected to node with `ID=1` and client `Beta` connected to node `ID=2` and they will both make requests for the resource.



(a) Node `ID=1`



(b) Node `ID=2`



(c) Node `ID=5`

Figure 12: Coordinator request queue in action.

As we can see from the screenshots above, the coordinator places the request from node `ID=2` into the `REQUEST_QUEUE` and it only sends it the `REQUEST_OK` message once it has received the `RELEASE_MESSAGE` from the node with `ID=1`, which had earlier acquired the lock on the resources.

In the following example we will show an example of the return operation carried out. The very first operation carried out was client Alpha borrowing book with `ID=1` and now we will attempt to return it back on shelf.

```
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
2
[CLIENT INTERFACE] Enter ID of the book you would like to return.
1
[2020-05-06 19:31:33 | Alpha ] Send to 1: RETURN UPDATE
[2020-05-06 19:31:33 | Alpha ] Receive from 1: Return Successful
```

(a) Client `ID=Alpha`

```
[ADMIN INTERFACE] Choose option: Ping Node(1), Election(2), Disconnect(3)
[2020-05-06 19:31:33 | 1 ] 172.20.10.3:61670 connected.
[2020-05-06 19:31:33 | 1 ] Receive from Alpha: RETURN UPDATE 1
[2020-05-06 19:31:33 | 1 ] Send to 5: REQUEST
[2020-05-06 19:31:33 | 1 ] 172.20.10.3:61672 connected.
[2020-05-06 19:31:33 | 1 ] Receive from 5: REQUEST OK 1
[RESOURCE] Returning book 1 from Alpha
[2020-05-06 19:31:33 | 1 ] Update Q - Book 1: Returned.
[2020-05-06 19:31:33 | 1 ] Send to 5: RELEASE
```

(b) Node `ID=1`

Figure 13: Returning a book borrowed earlier.

As we can see the node carries out the the appropriate procedures and the book is successfully returned on shelf.

In the last two examples we will validate the library access methods. The tests will be as follows:

- Return a book that is on shelf.
- Return a book held by another customer.
- Borrow a book that is held by another customer

As it stands book with ID=2 is held by customer Alpha, and book with ID=4 is held by customer Beta.

The following is when customer Alpha attempts to carry out all of the above, in series:

```
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
2
[CLIENT INTERFACE] Enter ID of the book you would like to return.
5
[2020-05-06 19:40:23 | Alpha ] Send to 1: RETURN UPDATE
[2020-05-06 19:40:23 | Alpha ] Receive from 1: Return Unsuccessful
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
2
[CLIENT INTERFACE] Enter ID of the book you would like to return.
4
[2020-05-06 19:40:30 | Alpha ] Send to 1: RETURN UPDATE
[2020-05-06 19:40:30 | Alpha ] Receive from 1: Return Unsuccessful
[CLIENT INTERFACE] Choose option: Borrow Book(1), Return Book(2), Disconnect(3)
1
[CLIENT INTERFACE] Enter ID of the book you would like to borrow.
4
[2020-05-06 19:40:36 | Alpha ] Send to 1: BORROW_UPDATE
[2020-05-06 19:40:36 | Alpha ] Receive from 1: Borrow Unsuccessful
```

(a) Client `ID=Alpha`

Figure 14: Validating library operations.

As we can see, none of these operations were successful and we can thus conclude that the system is operating correctly.

To verify the state of the library record, I have used the `print_list_details` of the `access_books.py` module whose output was:

```
>>> from access_books import *
>>> print_list_details()
book with ID: 1 Is on Loan? False It is held by:
book with ID: 2 Is on Loan? True It is held by: Alpha
book with ID: 3 Is on Loan? False It is held by:
book with ID: 4 Is on Loan? True It is held by: Beta
book with ID: 5 Is on Loan? False It is held by:
book with ID: 6 Is on Loan? False It is held by:
book with ID: 7 Is on Loan? False It is held by:
book with ID: 8 Is on Loan? False It is held by:
book with ID: 9 Is on Loan? False It is held by:
book with ID: 10 Is on Loan? False It is held by:
```

Figure 15: Library record state.

5 Evaluation

Overall I am very pleased with the result of my work as I have successfully implemented a distributed system that supports all of the functionality within the specification, with the addition of an leader election extension requirement. I am also very happy with the design of my system as it employs multi-threading which in theory means that it can handle a large number of clients connecting to it. Furthermore, the message protocol I have introduced allows for a very smooth intra-system communication between nodes as well as node-client communication. This in combination with the modularised code make the system highly maintainable and scalable as introducing additional features is as easy as adding new message types, and adding new methods that carry out the additional functionality. What I'm most proud of however, is the asynchronous nature of the resource access mechanism that I've implemented. In addition to the queue maintained by the coordinator, having a queue with outstanding client requests/operations in each node, the nodes can carry on with other work until they are granted access to the resource, maximising efficiency - they do not just busy wait until they are given the lock on the resource.

If I had more time, I would attempt to extend the functionality of the system by supporting joining nodes, meaning allowing the system to grow on demand, for example, when there is a high volume of communication. Lastly, I would've attempted to make the library records more sophisticated, perhaps by adding a database, and introducing other fields such as due date as well as functionality to allow clients to renew/extend their book loan.

6 Running

I have included a `README` file within the submission which provides comprehensive instructions on how to set up and run the distributed system implemented. If you wish to test the system please take good care to follow the instructions closely, and please make sure that the correct hostname values are passed within the `nodes.csv` file.

In the submission I have also included an examples of logs within the `logs` directory. However, note that I have wiped the records of the `book_list` resource prior to submission and so currently all of the books will be 'on shelf'.

References

[1] Toniolo, A. (2020). CS4013 (L12 W6): Coordintion - Part 2 Slide 89. https://studres.cs.st-andrews.ac.uk/CS4103/Lectures/L12_w6.pdf