

CSCI B 534 Distributed Systems

Assignment 2: Total Order Broadcast

Abstract:

In this Assignment, we aim to implement the Total Order Broadcast algorithm by developing a system consisting of many processes sending messages to each other and the order of message delivery is the same on all the processes (Total Order). We also make sure that the system follows FIFO (First In First Out) principle and a message with lower timestamp must be delivered before the message with a higher timestamp.

Architecture:

The driver program reads from the config file and spawns the specified number of processes. Each process creates an instance of the MyProcess class which is an abstraction for a process.

MyProcess:

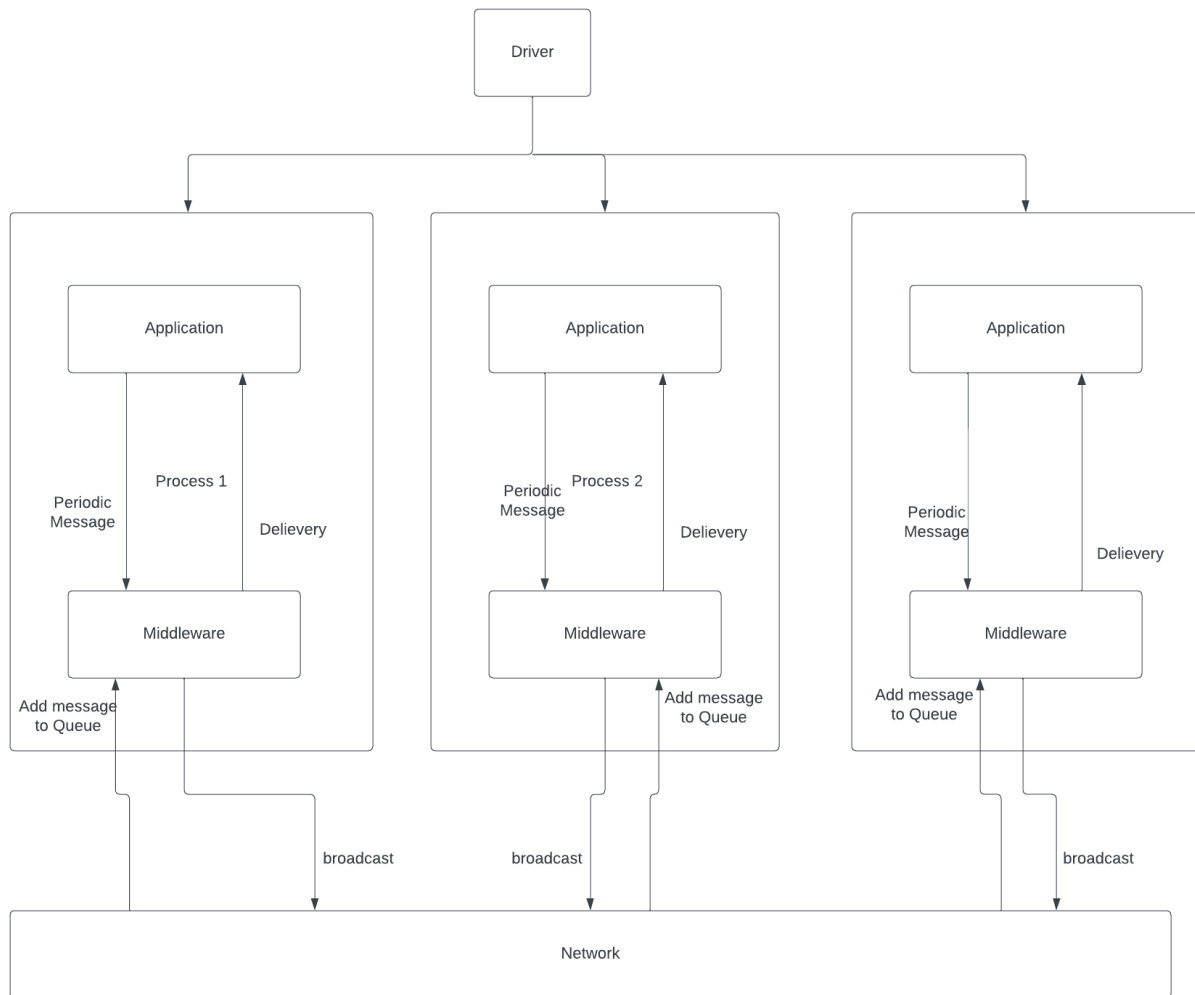
1. The MyProcess constructor in turn creates instances of Application and Middleware classes on separate threads.
2. Thus each Process has it's Application and Middleware instances. All the required ports are passed to the Application and Middleware from driver through the MyProcess instance.

Application:

1. The Application periodically sends messages to the Middleware using specified port.
2. The Application also listens for incoming messages from the Middleware. Both of these operations are concurrent and run on different threads.

Middleware:

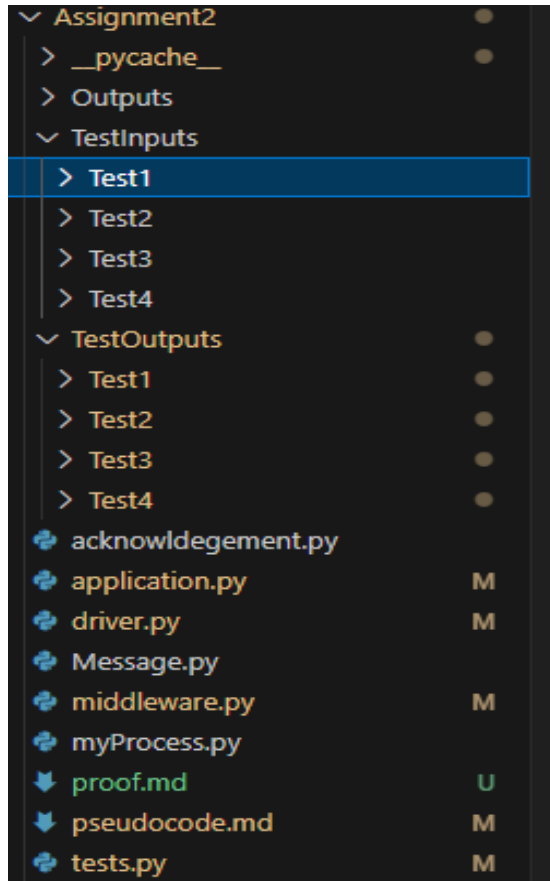
1. The Middleware has 3 concurrent threads for receiving messages from Application, for receiving messages from the network, and for processing of the message queue.
2. The Middleware maintains its local clock and this clock value is updated when messages are received from the Application and from the network.
3. When the Middleware receives a message from the Application, it increments its clock value by adding its own pid to the initial clock value. The Middleware then creates Message class object from the received message and then broadcasts the new message object throughout the network.
4. This Message object has a unique hash value and field called acks which keeps a track of the number of acknowledgements received.
5. When a process receives a message, if the message is an instance of the Message class that is it is an original message sent by some process, it pushes the message into the local queue.
6. This local queue is a priority queue or a min heap that has the message with the lowest timestamp at the top.
7. If the received message was an instance of the Acknowledgement class, the acknowledgment is mapped to its corresponding message using the unique hash.
8. The processing of the queue runs concurrently and checks whether the message at the top of the queue has received all the acknowledgements. If it has, the middleware delivers the message to the Application and pops that message from the queue. Otherwise it broadcasts the acknowledgement over the network.



File Structure

There is a folder called TestInputs which has folders for individual test case inputs. Each test case has a process_mapping.json and a network_receive.csv. The process_mapping.csv is a config file and has the necessary metadata like the ports to send and receive data from, Application and Middleware hostnames etc. The network_receive.csv file has a mapping between the process pid and the port over which it receives messages over the network. This is required for broadcasting the messages over the network. The corresponding outputs are stored in a folder called TestOutputs which has output folders for each test case. The driver.py is the starting

point of the application and is responsible for creating and terminating the processes. There is a file called tests.py which is a file for unit testing the code. It is called within the driver program.



Design Specifications:

The Application has a field called messages and a message counter. The messages is the list of capital letters from 'A' to 'Z'. The message counter is set to 0 when the application is spawned. The Application periodically creates a message by taking the message at the index corresponding to the message counter and add its pid to the message and then increments the message counter. The message counter is then mod

by 26 to ensure that the index does not go out of bounds. The Application sleeps for a random time between 1 to 5 seconds before sending the message.

Once this message is intercepted by the middleware, the middleware updates its local clock value by adding the pid to the current clock value.

Note: I tried updating the clock by incrementing the clock value by 1, but I received some out of order messages for a process during each run.

The received message is converted into an instance of the Message class and then broadcast over the network.

The Message class consists of the process pid, data_block which is the original message component that the Application sent, clock value, acks and a unique hash which is composed by applying the hash function over the combined string representation of the pid, data_block and the clock.

The Acknowledgement class has the same fields except the acks attribute. The unique hash attribute is used to map a particular message with its acknowledgement.

When broadcasting a message of the Message class over the network, the serialize method of the Message class is called which prepends the prefix 'MSG' and in the same way, message of the Acknowledgement class prepends the prefix 'ACK' in its serialize method.

When a process receives a message, the middleware checks if the message is a Message or an Acknowledgment message. If the message is an instance of the Message class, it updates the clock value by taking the max of the local clock and the clock of the received message and adds the process's pid to it. The message is then put in the local queue which is an implementation of the heapq (priority queue or min heap).

If the received message was an Acknowledgement, then the acknowledgement is pushed into a acknowledgement list and then I check that which particular messages have their acknowledgements in the acknowledgement list. When a mapping is found, the acks attribute of that message is incremented by 1.

The processing of the queue runs concurrently on a separate thread. If the message at the front of the queue has received the acknowledgements from all the processes, then this message is delivered to the Application and the message is popped from the queue.

Otherwise, an acknowledgement is broadcasted over the network. To avoid repeated acknowledgements, I store the hash of the sent acknowledgement in a dictionary and set the value to True. When trying to send the acknowledgement again, I check if the entry exists in the dictionary and if it does, I do not send the acknowledgement.

Running Instruction:

Run the driver.py from the command line by running the following command

```
python ./driver.py
```

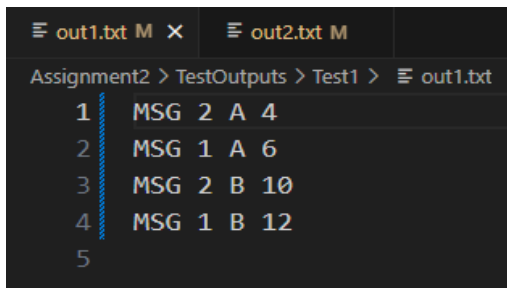
Note: The driver file runs each test case for 45 sec and the proceeds to the next one. The message processing order for each process is written in a file called outi.txt. Here i denotes the process id of the process. For instance, message processing order for process with pid 1 will be written in a file called out1.txt inside the specific test case subfolder.

Test Cases:

Test Case 1:

The format of the message is MSG pid data_block clock

Out1.txt



```
1 MSG 2 A 4
2 MSG 1 A 6
3 MSG 2 B 10
4 MSG 1 B 12
5
```

Out2.txt

```
...  out1.txt M  out2.txt M X
Assignment2 > TestOutputs > Test1 > out2.txt
1 MSG 2 A 4
2 MSG 1 A 6
3 MSG 2 B 10
4 MSG 1 B 12
5
```

Test Case 2:

Out1.txt

```
out1.txt M X  out2.txt M  out3.txt M
Assignment2 > TestOutputs > Test2 > out1.txt
1 MSG 3 A 6
2 MSG 2 A 10
3 MSG 1 A 12
4 MSG 2 B 14
5
```

Out2.txt

```
out1.txt M  out2.txt M X  out3.txt M
Assignment2 > TestOutputs > Test2 > out2.txt
1 MSG 3 A 6
2 MSG 2 A 10
3 MSG 1 A 12
4 MSG 2 B 14
5
```

Out3.txt

```
≡ out1.txt M    ≡ out2.txt M    ≡ out3.txt M X
Assignment2 > TestOutputs > Test2 > ≡ out3.txt
1  MSG 3 A 6
2  MSG 2 A 10
3  MSG 1 A 12
4  MSG 2 B 14
5  |
```

Test Case 3:

Out1.txt

```
≡ out1.txt M X    ≡ out2.txt M    ≡ out3.txt M    ≡ out4.txt M
Assignment2 > TestOutputs > Test3 > ≡ out1.txt
1  MSG 3 A 6
2  MSG 2 A 10
3  MSG 1 A 12
4  MSG 4 A 18
5  |
```

Out2.txt

```
≡ out1.txt M    ≡ out2.txt M X    ≡ out3.txt M    ≡ out4.txt M
Assignment2 > TestOutputs > Test3 > ≡ out2.txt
1  MSG 3 A 6
2  MSG 2 A 10
3  MSG 1 A 12
4  MSG 4 A 18
5  |
```

Out3.txt

```
≡ out1.txt M    ≡ out2.txt M    ≡ out3.txt M X    ≡ out4.txt M
Assignment2 > TestOutputs > Test3 > ≡ out3.txt
1  MSG 3 A 6
2  MSG 2 A 10
3  MSG 1 A 12
4  MSG 4 A 18
5  |
```


Out4.txt

```
Assignment2 > TestOutputs > Test3 > out4.txt
1 MSG 3 A 6
2 MSG 2 A 10
3 MSG 1 A 12
4 MSG 4 A 18
5
```

Test Case 4:

Out1.txt

```
Assignment2 > TestOutputs > Test4 > out1.txt
1 MSG 1 A 2
2 MSG 2 A 4
3 MSG 5 A 10
4 MSG 5 B 30
```

Out2.txt

```
Assignment2 > TestOutputs > Test4 > out2.txt
1 MSG 1 A 2
2 MSG 2 A 4
3 MSG 5 A 10
4 MSG 5 B 30
```

Out3.txt

```
Assignment2 > TestOutputs > Test4 > out3.txt
1 MSG 1 A 2
2 MSG 2 A 4
3 MSG 5 A 10
4 MSG 5 B 30
```

Out4.txt

```
Assignment2 > TestOutputs > Test4 > out4.txt
1 MSG 1 A 2
2 MSG 2 A 4
3 MSG 5 A 10
4 MSG 5 B 30
```

Out5.txt

```
Assignment2 > TestOutputs > Test4 > out5.txt
1 MSG 1 A 2
2 MSG 2 A 4
3 MSG 5 A 10
4 MSG 5 B 30
```

Known Limitations:

1. It is difficult to test correctness of total order as the number of processes increase, because multiprocessing depends on the number of cores a machine has and is limited by that.
2. As the number of processes increase, the inter process communication increases and creates additional communication overhead and introduces latency.
3. I have assigned the ports for sending and receiving the data in the config file in a random manner which is not ideal.

4. If a process crashes, the algorithm would come to a halt due to missing acknowledgements from that particular process.
5. Checking for correctness of Total Order by writing the message ordering to an output file may not be the best way.

Potential Improvements:

1. Ensure fault tolerance by allowing message retransmission in case of a process failure, so that the system does not come to a halt.
2. We can enforce a well defined network communication protocol that ensures reliability and consistency.
3. The port configuration should be done dynamically instead of reading from a config file.
4. The termination of the broadcast can be handled more effectively and gracefully by making sure that all resources are released.
5. More rigorous testing for scalability can be done.

References:

1. <https://docs.python.org/3/library/socket.html>
2. <https://docs.python.org/3/library/threading.html>
3. <https://docs.python.org/3/library/multiprocessing.html>
4. <https://lamport.azurewebsites.net/pubs/time-clocks.pdf>