P434: Report 2

Joshua Elms - Due 29 Feb. 2024

1 Pseudo-Code

```
def total_broadcast(msg):
    increment logical clock
    if msg is not an ack:
        add message to own message queue # instead of sending message
        sort message queue
    for pid in party:
        # don't send to self, we'll handle that behavior when receiving app message
        if self_pid != pid:
            create socket
            connect to pid # pid is port here
            send encoded msg over socket
            close socket

def receive_msg()
    listen on a socket
    receive message once connected
    decode message # I implemented a message class to handle encoding and decoding
    if msg is not from application: # app msgs don't have timestamps
        if msg timestamp > self timestamp:
            self timestamp = msg timestamp
        elif msg timestamp == self timestamp:
            # append self pid to self timestamp to solve timestamp collision
            self timestamp = self timestamp + (self pid / 10**ceil(log(10(self pid)))

    increment logical clock

    if msg is an ack:
        extract msg PID, TS and ack PID, TS
        if acked msg already in queue:
            add msg PID to ack list

        else:
            create ack list for acked msg # this happens if ack arrives before msg,
                                            common

        if msg is at head of your queue and has all acks:
            deliver msg to app
            increment logical clock
            dequeue msg

    elif msg is from app:
        broadcast with PID and TS to all party members
        sort msg queue by TS

    else: # new message from another process
        add msg to msg queue
        sort msg queue by TS
        if ack list for msg exists:
            add msg PID to ack list

        else:
            build ack list for msg, add msg PID

        broadcast ack with self PID and TS to all
        if msg is at head of your queue and has all acks:
            deliver msg to app
            increment logical clock
            dequeue msg

def deliver_msg(msg):
    print(f"Delivering msg:{msg} to application")
```

2 Proof of Correctness

2.1 Assumptions

a) Proving total order between any two elements is sufficient to claim the algorithm's total order.

b) FIFO: if process A sends two messages to process B, they will arrive in the same order as they were sent.

2.2 Steps

We will prove by contradiction that the order in which messages are delivered to their applications will be shared by all processes.

1. Let process A deliver message X at (logical) time i and process B deliver message Y at time j, where i < j.
2. For B to deliver message j:Y, it would have to have j:Y at the front of its queue and received all acks.
3. If j:Y is at the front of B's sorted queue, then i:X has not been received by B.
4. If i:X has not been received by B, then by the FIFO property, it could not have received A's ack of j:Y, which must have been sent after i:X.
5. This violates the requirement that a message must have all acks in order to be delivered, therefore such a situation could not occur.

3 Implementation Details

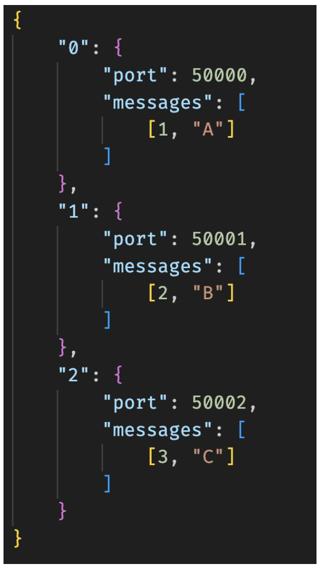
3.1 Architecture

My implementation is object-oriented. I have created four classes: Message, MessageQueue, Process, and Application. Process contains a MessageQueue which itself contains both Messages and a list of acknowledgements of those Messages. The Application class is solely to pass Messages to the Process (middleware) for broadcast and eventual delivery back to applications.

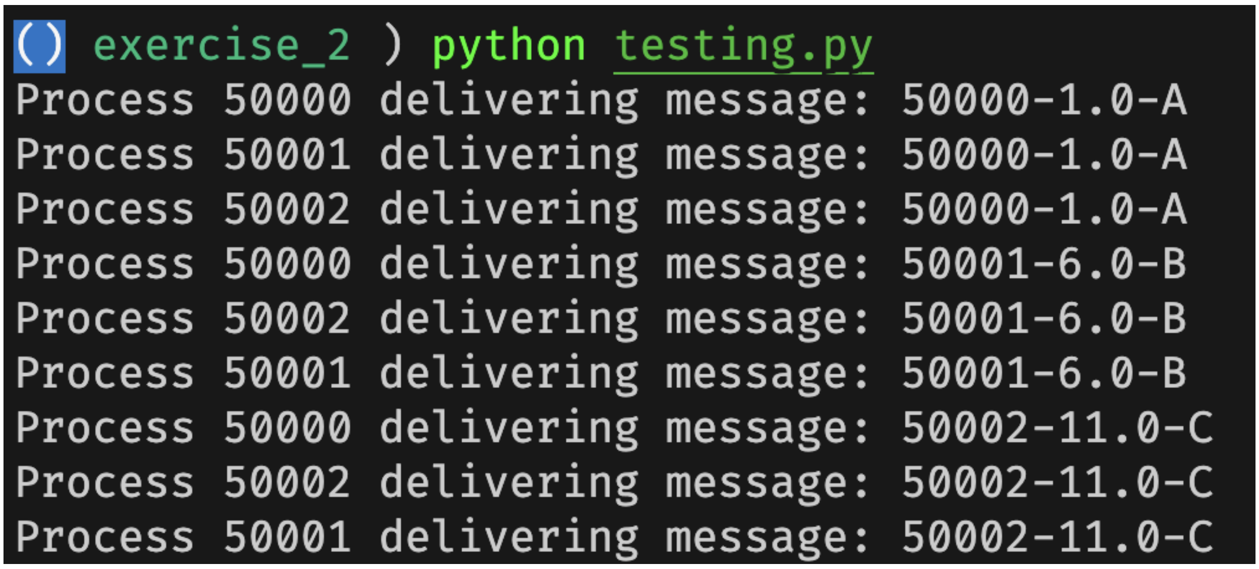
3.2 Tests

I have included three tests here, but generating new tests is easy with my test suite. In the following sets of images, the first shows the test input and the second shows the output produces. The input files include a port number for each process (which doubles as process ID) and messages for the application to deliver to the process. Each message takes the form [request time by app, message content]. The message format in the output image is port-lamport_time-message.

1. Test 1: 3 applications make requests serially, a second apart. Outputs are ordered as expected.

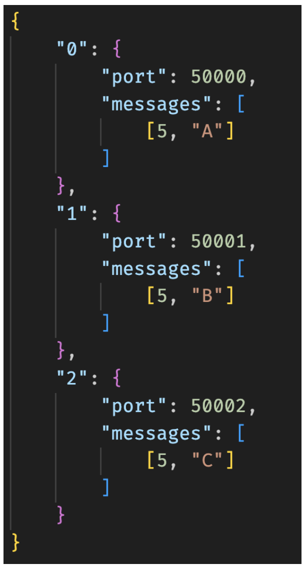


```
{
    "0": {
        "port": 50000,
        "messages": [
            [1, "A"]
        ]
    },
    "1": {
        "port": 50001,
        "messages": [
            [2, "B"]
        ]
    },
    "2": {
        "port": 50002,
        "messages": [
            [3, "C"]
        ]
    }
}
```

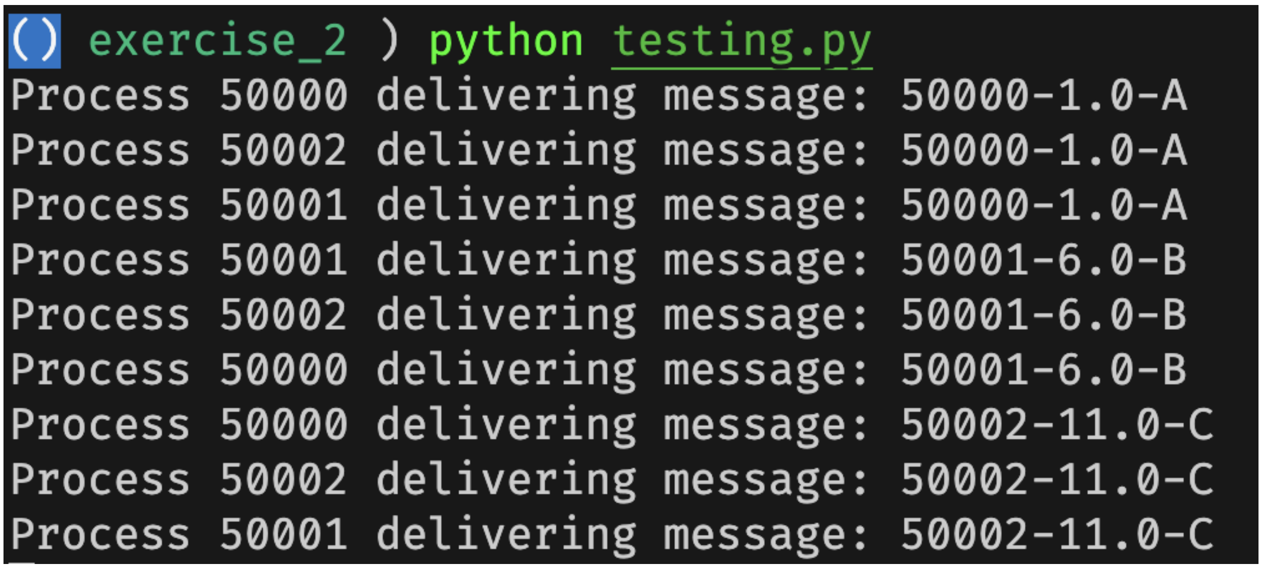


```
() exercise_2 ) python testing.py
Process 50000 delivering message: 50000-1.0-A
Process 50001 delivering message: 50000-1.0-A
Process 50002 delivering message: 50000-1.0-A
Process 50000 delivering message: 50001-6.0-B
Process 50002 delivering message: 50001-6.0-B
Process 50001 delivering message: 50001-6.0-B
Process 50000 delivering message: 50002-11.0-C
Process 50002 delivering message: 50002-11.0-C
Process 50001 delivering message: 50002-11.0-C
```

2. Test 2: 3 applications all put in requests to the middleware at precisely the same time. Outputs are coincidentally ordered “A”, “B”, “C”, but any order would be acceptable.

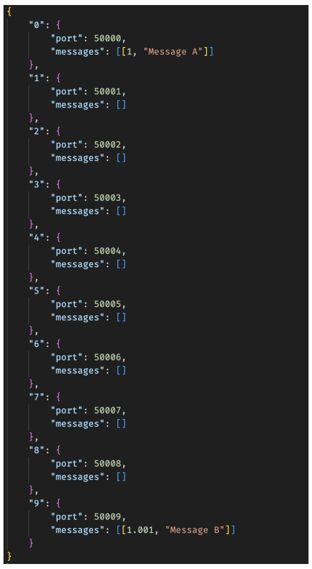


```
{
    "0": {
        "port": 50000,
        "messages": [
            [5, "A"]
        ]
    },
    "1": {
        "port": 50001,
        "messages": [
            [5, "B"]
        ]
    },
    "2": {
        "port": 50002,
        "messages": [
            [5, "C"]
        ]
    }
}
```

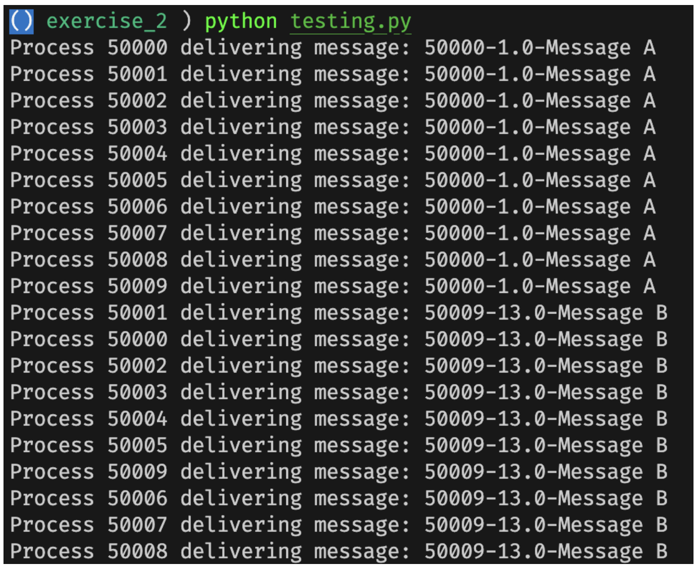


```
() exercise_2 ) python testing.py
Process 50000 delivering message: 50000-1.0-A
Process 50002 delivering message: 50000-1.0-A
Process 50001 delivering message: 50000-1.0-A
Process 50001 delivering message: 50001-6.0-B
Process 50002 delivering message: 50001-6.0-B
Process 50000 delivering message: 50001-6.0-B
Process 50000 delivering message: 50002-11.0-C
Process 50002 delivering message: 50002-11.0-C
Process 50001 delivering message: 50002-11.0-C
```

3. Test 3: 2 applications put in serial requests to a network of 10 processes, which handle deliver the messages in the proper order.



```
{
    "0": {
        "port": 50000,
        "messages": [[1, "Message A"]]
    },
    "1": {
        "port": 50001,
        "messages": []
    },
    "2": {
        "port": 50002,
        "messages": []
    },
    "3": {
        "port": 50003,
        "messages": []
    },
    "4": {
        "port": 50004,
        "messages": []
    },
    "5": {
        "port": 50005,
        "messages": []
    },
    "6": {
        "port": 50006,
        "messages": []
    },
    "7": {
        "port": 50007,
        "messages": []
    },
    "8": {
        "port": 50008,
        "messages": []
    },
    "9": {
        "port": 50009,
        "messages": [[1.001, "Message B"]]
    }
}
```



```
() exercise_2 ) python testing.py
Process 50000 delivering message: 50000-1.0-Message A
Process 50001 delivering message: 50000-1.0-Message A
Process 50002 delivering message: 50000-1.0-Message A
Process 50003 delivering message: 50000-1.0-Message A
Process 50004 delivering message: 50000-1.0-Message A
Process 50005 delivering message: 50000-1.0-Message A
Process 50006 delivering message: 50000-1.0-Message A
Process 50007 delivering message: 50000-1.0-Message A
Process 50008 delivering message: 50000-1.0-Message A
Process 50009 delivering message: 50000-1.0-Message A
Process 50001 delivering message: 50009-13.0-Message B
Process 50000 delivering message: 50009-13.0-Message B
Process 50002 delivering message: 50009-13.0-Message B
Process 50003 delivering message: 50009-13.0-Message B
Process 50004 delivering message: 50009-13.0-Message B
Process 50005 delivering message: 50009-13.0-Message B
Process 50009 delivering message: 50009-13.0-Message B
Process 50006 delivering message: 50009-13.0-Message B
Process 50007 delivering message: 50009-13.0-Message B
Process 50008 delivering message: 50009-13.0-Message B
```

3.3 Possible Improvements

1. While I currently deliver messages to applications with a simple print statement, I could have each application open up a socket to listen on and send responses from the middleware.
2. As in the previous assignment, I have taken a single-threaded approach: the processes are each only able to handle communicating with one other process at a time. This is a non-issue here, as they have a large backlog and can quickly make their ways through the other requests if they ever pile up, but for production applications this may be a problem.
3. Many small optimizations could be made; not attempting to deliver the message on each attempt, but rather when a certain number of acks had been recorded, etc. I do not consider these optimizations to be relevant to this class, though, and the system works perfectly well.

4 References

1. Assignment 2: Totally Ordered Multicast
2. json.dump() in Python
3. Socket Programming in Python
4. socket - Low-level networking interface