Pseudo Code fo Total Order BroadCast

Overview

The algorithm has it's entry point in a file called driver.py which reads from a json nconfig file that has data related to number of processes and diferent ports used by the processes. The driver spawns n processes where n is the number of json configurations in the config file. Each process creates an instance of MyProcess class which is an abstraction for a process. The MyProcess class object spawns two threads, one to instantiate an Application instance and the other to instantiate a Middleware instance.

The Application listens for incoming messages from Middleware on a port specified in the config file. The Application periodically sends some messages to the Middleware in a specific format. This message is an instance of the Message class. The clock is adjusted here by incrementing it's value by 1. The Middleware has 2 ports that are specified in the config file, one for receiving messages from the Application and the other for receiving messages from the Network. The Middleware in turn sends this message to the Network by broadcasting it. All the processes then adjsut their clock by updating the clock value to the max of local clock and clock of the received message and add this message to their local queue. The Middleware maintains a record of acknowledgement's that it has sent. When the sender process has received all the acknowledgements for it's message which is at the front of the message queue, it delievers the message to the Application layer. Receiving messages from Application, receiving messages from Network and queue processing all happen on separate threads.

Application

```
def sendRequestToMiddleware():
    while True:
        # send the message to Middleware
        # messgaes is a list of capital letters from A to Z
        sleep(randomInt(1, 5))
        message = messages[msg_counter] + pid
        msg_counter = (msg_counter + 1) % 26
        #send the data to Middleware by establishing the socket connection
        middleware_socket.connect(middleware_host, middleware_port)
        middleware_socket.send(message)



def receiveFromMiddleware():
    #Bind the socket and start listening
    app_socket.bind(app_host, app_port)
    while True:
        conn, addr = socket.accept()
        data = conn.receive()

        #writet the data to file
        writeToOutputFile(data)


def run():
    Thread(receivedMessage).start()
```

Middleware

```
def receiveFromApplication():
    middleware_socket.bind(host, middleware_application_port)
    while True:
        conn, addr = socket.accept()
        data = conn.receive()

       clock = clock + pid   #Increment the local clock value by 1
       message = Message(data)  #Create the Message Object
       sendToNetwork(message)  #Broadcast the message to all the processes

def read_network_ports():
    # Read the network_receive.csv file and map the network port for each process with the process pid
    network_ports[pid] = port
```

```
def sendToNetwork(message):
    read_network_ports()   #Iterate over network ports from config file for the broadcast
    to_network_socket.connect(middleware_host, network_ports[pid])
    to_network_socket.send(message)   #Broadcast the message to all the processes

def processAcks():
    for msg in queue:
        for ack in ack_list:
            map_msg_to_ack()   # Map each ack received to it's corresponding msg


def receiveFromNetwork():
    from_network_socket.bind(host, middleware_network_port)
    while True:
        conn, addr = socket.accept()
        data = conn.receive()

        if type data is MSG:
            msg = Message.deserialize(data)
            clock = max(clock, msg.clock) + pid  #Update the clock by taking max of local clock and clock of the r
            queue.push(msg)  #Push the msg into the local queue (priority queue or heap)

        else:
            ackObj = Acknowledgement.deserialize(data)
            ack_list.push(ackObj)
            processAcks()   # call the function to process Acknowledgement Object

def sendToApplication(message):
    to_application_socket.connect(host, application_middleware_port)
    to_application_socket.send(message)   #Broadcast the message to all the processes


def processQueue():
    if queue.front().acks is 0:
        sendToApplication(queue.front())  #Delever the message to the Application
        queue.pop()   #Remove the message from the queue

    else:
        if queue.front().hash not in ack_dict:
            sendToNetwork(Acknowledgement.deserialize(queue.front()))   #Broadcast the acknowledgement over the ne
            add_ack_to_dict()   #Record the entry for the ack in the local dictionary


def run():
    Thread(receiveFromApplication).start()
    Thread(receiveFromNetwork).start()
    Thread(processQueue).start()
```

Message

```
#Message class has the following attributes
# 1. pid
# 2. data_block
# 3. clock
# 4. hash = hash(pid + data_block + clock)
# 5. acks
def serialize():
    return "MSG " + pid + data_block + clock

@staticmethod
def deserialize(msg):
    msg = Message(msg.split(" "))   #Create the Message object from the received message
```

Acknowledgement

```
#Acknowledgement class has the following attributes
# 1. pid
# 2. data_block
# 3. clock
# 4. hash = hash(pid + data_block + clock)
def serialize():
    return "ACK " + pid + data_block + clock

@staticmethod
def deserialize(msg):
    msg = Acknowledgement(msg.split(" "))   #Create the Acknowledgement object from the received message
```

MyProcess (Abstraction for Process)

```
def spawn_Application():
    Thread(Application()).start()


def spawn_Middleware():
    Thread(Middleware()).start()
```