

CSE 5306
Distributed Systems
Fall-2019
PROJECT – 2

NAME: Zahidur Rahim Talukder

ID: 1001771748

NAME: Pranjol Sen Gupta

ID: 1001763076

I have neither given or received unauthorized assistance on this work

Signed:

Date: 11/17/2019

Pranjol Sen Gupta,

Zahidur Rahim Talukder

ASSIGNMENT-1

Dummy_process1.py, Dummy_process2.py, Dummy_process3.py

In this project, we have implemented totally ordered multicasting using Lamport's algorithm. Each process conducts local operations and numbers them as PID.EVENT_ID. After each operation is done, a process multicast the event to all other processes in the distributed system. The expected outcome of this assignment is that events occurred at different processes will appear in the same order at each individual process. To realize such a total order of events, each process maintains a buffer for received events and we have followed the rules on slide 19 in Lecture 6 when delivering the events. In this assignment, the delivery of events is simply printing them on screen, in the format of CURRENT_PID: PID.EVENT_ID. We have used three threads in each process to handle the communication, processing and deal with message delivery, respectively.

The address of multicast group is "127.0.0.1" and multicast_ports are = [6001, 6002, 6003].

Each process uses different port. We have used three processes in this project. Each process has three threads which are `rcv_thread = ReceiverThread()`, `send_thread = EventThread()`, `process_thread = ProcessingThread()`. Each thread inherits from the base thread class. In each sub class, the run method has been overridden.

We have defined a Test class in each process, which consists of process id and logical clock of each process.

At first each process checks its own receiving thread whether there is any multicast event from other process. If there is any event it puts them into queue by using `event_queue.put(event)`. Before putting them in the queue, it deserialize the object using `event = pickle.loads(data)`.

In the processing thread events are fetched from queue, using `event = event_queue.get()`. Based on the Lamport's algorithm events are then printed in the screen by using:

```
print ("the event is 1.{}".format(v[i]))
```

In the event thread, the process updates its own event and multicast it to all other processes using `sock.sendto(x, (multicast_group, port))`. Each process has 10 events. Before sending the event to all other processes including the process itself, the event is serialized using `x = pickle.dumps(event)`.

Output: From the following figure we can see that all the processes follow Lamport's algorithm.

Process 1:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/dummy_multicast/Dummy_process1.py"
send event, updating event 1.1
send event, updating event 1.2
the received event is 3.1
send event, updating event 1.3
the received event is 2.1
send event, updating event 1.4
the received event is 3.2
the received event is 2.2
send event, updating event 1.5
the received event is 3.3
send event, updating event 1.6
the received event is 2.3
send event, updating event 1.7
the received event is 3.4
the received event is 2.4
send event, updating event 1.8
send event, updating event 1.9
the received event is 2.5
the received event is 3.5
send event, updating event 1.10
the received event is 3.6
the received event is 2.6
the received event is 3.7
the received event is 2.7
the received event is 2.8
the received event is 3.8
the received event is 2.9
the received event is 3.9
the received event is 2.10
the received event is 3.10
1
```

Process 2:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/dummy_multicast/Dummy_process2.py"
the received event is 1.2
the received event is 3.1
the received event is 1.3
send event, updating event 2.1
the received event is 1.4
the received event is 3.2
send event, updating event 2.2
the received event is 1.5
the received event is 3.3
the received event is 1.6
send event, updating event 2.3
the received event is 1.7
the received event is 3.4
send event, updating event 2.4
the received event is 1.8
the received event is 1.9
send event, updating event 2.5
the received event is 3.5
the received event is 1.10
the received event is 3.6
send event, updating event 2.6
the received event is 3.7
send event, updating event 2.7
send event, updating event 2.8
the received event is 3.8
send event, updating event 2.9
the received event is 3.9
send event, updating event 2.10
the received event is 3.10
```

Process 3:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/dummy_multicast/Dummy_process3.py"
the received event is 1.1
the received event is 1.2
send event, updating event 3.1
the received event is 1.3
the received event is 2.1
the received event is 1.4
send event, updating event 3.2
the received event is 2.2
the received event is 1.5
send event, updating event 3.3
the received event is 1.6
the received event is 2.3
the received event is 1.7
send event, updating event 3.4
the received event is 2.4
the received event is 1.8
the received event is 1.9
the received event is 2.5
send event, updating event 3.5
the received event is 1.10
send event, updating event 3.6
the received event is 2.6
send event, updating event 3.7
the received event is 2.7
the received event is 2.8
send event, updating event 3.8
the received event is 2.9
send event, updating event 3.9
the received event is 2.10
send event, updating event 3.10
```

ASSIGNMENT-2

Dummy_vec_process1.py, Dummy_vec_process2.py, Dummy_vec_process3.py

In this project, we have implemented the vector clock algorithm to enable totally ordered events in the distributed system. Similar to the previous assignment, we have used multiple processes to emulate multiple nodes. We have assumed that all nodes are initiated to the same vector clock. After completing a local operation, each process sends its updated vector clock to all other processes. We have followed the steps on slide 34 in Lecture 6 to implement the vector clock algorithm.

We have used three processes for this project. The address group is "127.0.0.1" and _ports are = [6001, 6002, 6003].

We have used two threads in each process. One is EventThread(Thread) and another is ReceiverThread(Thread). Both threads inherit from base class Thread. In each sub class, the run method has been overridden.

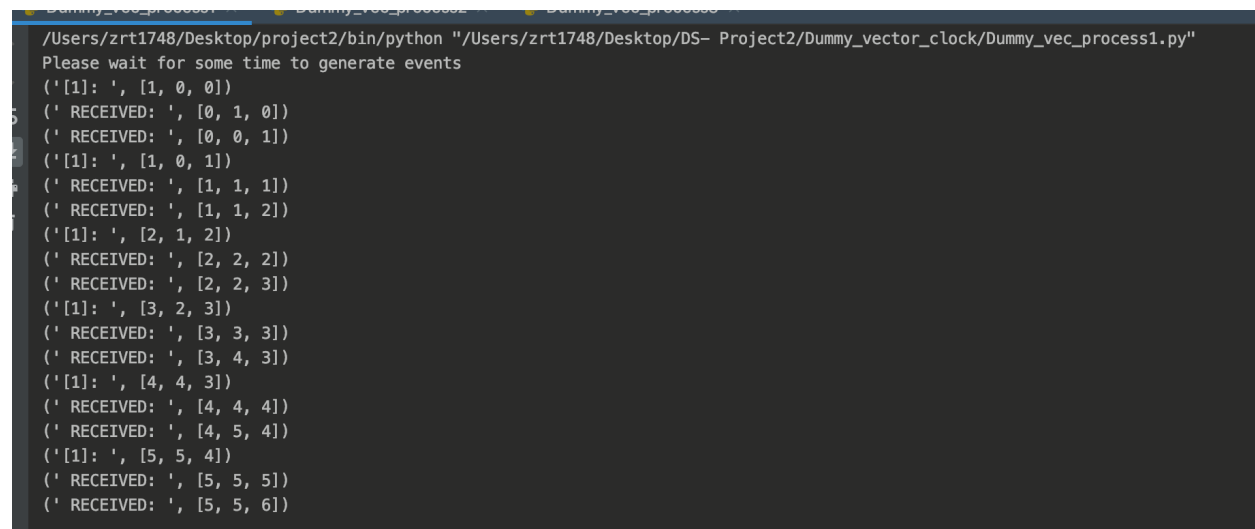
Each process has its own clock and process id.

In receiving thread, the thread simply checks for any incoming event and if there is any event, it prints them on screen.

In event thread the thread prints its own event and sends it to other processes, except it.

From the following figure we can see the output of vector clock for three processes.

Process1:



```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/Dummy_vector_clock/Dummy_vec_process1.py"
Please wait for some time to generate events
('1': '[1, 0, 0]')
(' RECEIVED: '[0, 1, 0]')
(' RECEIVED: '[0, 0, 1]')
('1': '[1, 0, 1]')
(' RECEIVED: '[1, 1, 1]')
(' RECEIVED: '[1, 1, 2]')
('1': '[2, 1, 2]')
(' RECEIVED: '[2, 2, 2]')
(' RECEIVED: '[2, 2, 3]')
('1': '[3, 2, 3]')
(' RECEIVED: '[3, 3, 3]')
(' RECEIVED: '[3, 4, 3]')
('1': '[4, 4, 3]')
(' RECEIVED: '[4, 4, 4]')
(' RECEIVED: '[4, 5, 4]')
('1': '[5, 5, 4]')
(' RECEIVED: '[5, 5, 5]')
(' RECEIVED: '[5, 5, 6]')
```

Process2:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/Dummy_vector_clock/Dummy_vec_process2.py"
Please wait for some time to generate events
('2]: ', [0, 1, 0])
('RECEIVED: ', [0, 0, 1])
('RECEIVED: ', [1, 0, 1])
('2]: ', [1, 1, 1])
('RECEIVED: ', [1, 1, 2])
('RECEIVED: ', [2, 1, 2])
('2]: ', [2, 2, 2])
('RECEIVED: ', [2, 2, 3])
('RECEIVED: ', [3, 2, 3])
('2]: ', [3, 3, 3])
('2]: ', [3, 4, 3])
('RECEIVED: ', [4, 4, 3])
('RECEIVED: ', [4, 4, 4])
('2]: ', [4, 5, 4])
('RECEIVED: ', [5, 5, 4])
('RECEIVED: ', [5, 5, 5])
('RECEIVED: ', [5, 5, 6])
```

Process3:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/Dummy_vector_clock/Dummy_vec_process3.py"
Please wait for some time to generate events
('3]: ', [0, 0, 1])
(' RECEIVED: ', [1, 0, 1])
(' RECEIVED: ', [1, 1, 1])
('3]: ', [1, 1, 2])
(' RECEIVED: ', [2, 1, 2])
(' RECEIVED: ', [2, 2, 2])
('3]: ', [2, 2, 3])
(' RECEIVED: ', [3, 2, 3])
(' RECEIVED: ', [3, 3, 3])
(' RECEIVED: ', [3, 4, 3])
(' RECEIVED: ', [4, 4, 3])
('3]: ', [4, 4, 4])
(' RECEIVED: ', [4, 5, 4])
(' RECEIVED: ', [5, 5, 4])
('3]: ', [5, 5, 5])
('3]: ', [5, 5, 6])
|
```

ASSIGNMENT 3

Centralize.py, Client1.py, Client2.py, Client3.py

In this project, we have implemented a locking scheme to protect a shared file in distributed system. While we use processes on a single machine to emulate the distributed system and there are shared-memory synchronization primitives available, we have implemented centralized distributed locking. When a process acquires the lock, it simply opens the file, increments a counter in the file, and closes the file. We have assumed that all processes keep requesting the lock until successfully acquiring the lock for a predefined number of times.

We have used different threads to process client request. Whenever a client is connected to the server, a thread starts, which is responsible for serving that particular client. If the client sends command “ACQUIRE”, the server will check at first if the lock status is true or not. If lock status is true, that means other client is busy with that file, the server rejects the requesting client and subsequently put the requesting client in a queue by using **process_queue.put((connection, address))**. If the current client is done with operating on the file, it sends “RELEASE” and the server raise the lock status to “FALSE”, that means another client is ready to operate on the file. Now if other client starts requesting to the sever, the server will check the queue first abd base on the queueing it will start serving the clients.

Here the challenging part was the implementation of the locking. We have used fcntl. Module; `fcntl.flock(lock_file, fcntl.LOCK_EX)` command to lock the file on the server and `fcntl.flock(lock_file, fcntl.LOCK_UN)` to unlock it.

On the client side, after a client gets lock, it opens the file, increments the value of the file and closes the file. We have used `file_obj.seek(0)` to keep the file pointer at the first position when we have tried to increment the file.

For this project we have used three clients who always try to get the lock and only one client gets it at a time. When one client gets the lock, it can solely increment the value of the file. So that the other cannot access the file.

From the following figure we can realize the implementation of the Centralized Distribution Locking System. Three processes are always trying to get the lock and try to increment the value in the test file. But at a time only one process gets the lock and increment the value.

Process1:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/problem 3/Client1.py"
Connecting to port 8080 and server 127.0.0.1
Connected to server and got acknowledgments: ACK
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                224
Requesting for access to the server...
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                228
Requesting for access to the server...
Requesting for access to the server...
Closing connect...

Process finished with exit code 0
|
```

Process2:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/problem 3/client2.py"
Connecting to port 8080 and server 127.0.0.1
Connected to server and got acknowledgments: ACK
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                225
Requesting for access to the server...
Requesting for access to the server...
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                227
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Requesting for access to the server...
Requesting for access to the server...
Access is denied by server. retry again..
Closing connect...

Process finished with exit code 0
```


Process3:

```
/Users/zrt1748/Desktop/project2/bin/python "/Users/zrt1748/Desktop/DS- Project2/problem 3/Client3.py"
Connecting to port 8080 and server 127.0.0.1
Connected to server and got acknowledgments: ACK
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                226
Requesting for access to the server...
Requesting for access to the server...
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Access is denied by server. retry again..
Requesting for access to the server...
Got Access to the file. Editing File...
    After editing, file content is:                229
Requesting for access to the server...
Access is denied by server. retry again..
Closing connect...

Process finished with exit code 0
```

Conclusion:

After doing the project we have learned the difference between Lamport and Vector clock. Although similar they have different purposes: vector clock can distinguish whether two operations are concurrent, or one is causally dependent on the other; Lamport timestamps enforces total ordering. Total ordering although more compact, cannot tell whether two operations are concurrent or causally dependent.

And from the third task we have learned how to implement the locking in the distribution system. The purpose of a lock is to ensure that among several nodes that might try to do the same piece of work, only one actually does it (at least only one at a time). That work might be to write some data to a shared storage system, to perform some computation, to call some external API, or suchlike. At a high level, there are two reasons why one might want a lock in a distributed application: for efficiency or for correctness.