Distributed System
(CSE431)

Project
On
Simulate message delivery guarantees such as Causal and Arbitrary, and their impact on some Mutual Exclusion distributed algorithm

Submitted to:
Prof. Lini Thomas

Submitted by:
Giridhari Lal Gupta (2018201019)
Varun Gupta (2018201003)
Anjul Gupta (2018201021)

Part 1
Simulate message delivery guarantees such as Causal and Arbitrary

In order to demonstrate this part of the project, we have implemented a distributed chat room which uses a ***fault tolerant causal channel*** to broadcast messages to every other client.

For implementing a causal channel algorithm, we have used ***Birman Schiper Stephenson protocol.***

In the project, we have assumed that there would be ***3 clients*** available.When a process receives a Message, it creates a new thread for each message. Each thread delays the message by a random time (**to simulate network delay**) and then compares its vector timestamp with the vector timestamp of the Message received. If it is not in accordance with the protocol, the message gets saved in the holdback queue and later on when its vector timestamp is in accordance with that of Message vector timestamp ,Message gets displayed.

During conversation, any of the clients can go down but still messages sent by others will reach them out when they are up. During the time they are not available, their message will get piled up in their holdback queue. And, when they are up, the holdback queue will be processed and messages will be received in causal fashion accordingly. This is how we have handled the Fault Tolerant part.

Birman-Schiper-Stephenson protocol

Introduction

The goal of this protocol is to preserve ordering in the sending of messages. For example, if *send*(m_1) -> *send*(m_2), then for all processes that receive both m_1 and m_2, *receive*(m_1) -> *receive*(m_2). The basic idea is that m_2 is not given to the process until m_1 is given. This means a buffer is needed for pending deliveries. Also, each message has an associated vector that contains information for the recipient to determine if another message preceded it. Also, we shall assume all messages are broadcast. Clocks are updated only when messages are sent.

Notation

- n processes
- P_i process
- C_i vector clock associated with process P_i; jth element is $C_i[j]$ and contains P_i's latest value for the current time in process P_j
- t^m vector timestamp for message m (stamped after local clock is incremented)

Protocol

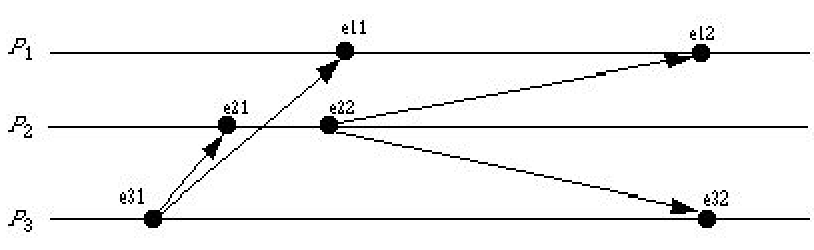
P_i* sends a message to *P_j

1. P_i increments $C_i[i]$ and sets the timestamp $t^m = C_i[i]$ for message m.

P_j* receives a message from *P_i

1. When P_j, j != i, receives m with timestamp t^m, it delays the message's delivery until both:
    1. $C_j[i] = t^m[i] - 1$; and
    2. for all $k <= n$ and k != i, $C_j[k] <= t^m[k]$.
2. When the message is delivered to P_j, update P_j's vector clock
3. Check buffered messages to see if any can be delivered.

Example



Here is the protocol applied to the above situation:

*e*31: P_3 sends message *a*; C_3 = (0, 0, 1); t^a = (0, 0, 1)

$e21$: P_2 receives message a. As $C_2 = (0, 0, 0)$, $C_2[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_2[1] => t^a[1]$ and $C_2[2] => t^a[2] = 0$. So the message is accepted, and C_2 is set to (0, 0, 1)

$e11$: P_1 receives message a. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_1[1] => t^a[1]$ and $C_1[2] => t^a[2] = 0$. So the message is accepted, and C_1 is set to (0, 0, 1)

$e22$: P_2 sends message b; $C_2 = (0, 1, 1)$; $t^b = (0, 1, 1)$

$e12$: P_1 receives message b. As $C_1 = (0, 0, 1)$, $C_1[2] = t^b[2] - 1 = 1- 1 = 0$ and $C_1[1] => t^b[1]$ and $C_1[3] =>$ tb[2] = 0. So the message is accepted, and C_1 is set to (0, 1, 1)

$e32$: P_3 receives message b. As $C_3 = (0, 0, 1)$, $C_3[2] = t^b[2] - 1 = 1 - 1 = 1$ and $C_1[1] => t^b[1]$ and $C_1[3] =>$ tb[2] = 0. So the message is accepted, and C_3 is set to (0, 1, 1)

Now, suppose t^a arrived as event $e12$, and t^b as event $e11$. Then the progression of time in P_1 goes like this:

$e11$: P_1 receives message b. As $C_1 = (0, 0, 0)$, $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$ and $C_1[1] => t^b[1]$, but $C_1[3] < t^b[3]$, so the message is held until another message arrives. The vector clock updating algorithm is not run.

$e12$: P_1 receives message a. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$, $C_1[1] => t^a[1]$, and $C_1[2] => t^a[2]$. The message is accepted and C_1 is set to (0, 0, 1). Now the queue is checked. As $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$, $C_1[1] => t^b[1]$, and $C_1[3] => t^b[3]$, that message is accepted and C_1 is set to (0, 1, 1).

Important Data Structures

Hold Back Queue

```
//datastructure of holdback_queue
struct holdmessage
{
    int sender;
    int p1;
    int p2;
    int p3;
    string message;
};

std::vector<holdmessage> holdback_queue;
```

This queue will store the messages when messages received are not in accordance with causal fashion property i.e when a particular message is received before some other expected message.

Each client has its own holdback queue. Holdback queue is basically a vector of holdmessage structure.

"***holdmessage***" structure comprises of:-

- Sender:- message sender client id(1,2 or 3)
- p1,p2,p3 :- vector timestamp values of the sender.
- Message:- message send by sender

Vector timestamp corresponding to each client

Each client has a globally available vector(of size 3) for storing timestamps of all three clients.

Working of Part 1

In order to implement causal ordering, we created a ***vector timestamp*** for each process. Every time a process multicasts a message out, it increments the number that corresponds to that process in its vector timestamp.

For every single process, the multicasting process sends an object ***CausalMessage***. A CausalMessage includes the message passed in, the vector timestamp of the multicasting process, the process ID of the multicasting process, and the MetaData of the destination process. When a process receives a CausalMessage, it creates a new thread for each message. Each thread delays the message by a random time (to simulate network delay) and then compares its vector timestamp with the vector timestamp of the CausalMessage (let's call it mesgTimes).

Both vectors grab the element at the index of the current processID. If mesgTimes's element is equal to current vector timestamp's element + 1, then we check if all of the rest of the elements of the vector timestamps are greater than the mesgTimes. If true, we deliver the message. If not, we determine if the events are concurrent by checking if each neither the current vector timestamp ≤ mesgTimes nor vice versa. If concurrent, we deliver the message. Otherwise, we store the message in the ***hold back queue***. Every time a message is delivered, we parse our hold back queue and determine if the messages queued are ready to be delivered by running the same algorithm on them. If true, we deliver them. Otherwise, we let them stay in the hold back queue.

Handling Fault Tolerance

When a process goes down, it stores its current vector timestamp in a file named, ***<processid>.txt*** . Any message sends to it will get stored in the file named, <***processid>_holdback.txt*** .

When the node is up, it will read back ***<processid>.txt*** and will accordingly updates its vector timestamp. Later, will look for the file <***processid>_holdback.txt*** and if its not empty, will read it and print the message in the causal fashion after processing using the above protocol.

Execution of part 1

- Firstly, start the server. In one of the command tab, run the below commands:-
  Compile the server code
  g++ server.cpp -pthread

  Run the server code
  ./a.out <server_port_number>

- Then, start the 3 clients
  Compile the client code
  g++ client.cpp -pthread

  Run the client
  ./a.out <process_number> <server_port_number>

  Client 1
  ./a.out 1 <server_port_number>

  Client 2
  ./a.out 2 <server_port_number>

  Client 3
  ./a.out 3 <server_port_number>

- Simulating ***message delivery guarantees (i.e Fault tolerance)***
  - All three clients are up and running.
  - On any client (let's say 3), type "***exit***" and ***press enter***. It's vector timestamp will be stored in a file named, "***3.txt***" .
  - Now, send messages from two other clients(1 and 2 in this case). You can notice the messages getting saved in ***3_holdback.txt*** .
  - Bring the client 3 up, using ./a.out 3 <server_port_number> . You will notice the messages sent by others(while client 3 was down) are getting printed in causal fashion.

Part 2

Distributed Mutual Exclusion Based on Causal Ordering

We have used ***Suzuki-Kasami's algorithm based on causal ordering (i.e Token Based Algorithm)*** for implementing distributed Mutual Exclusion. Token-based algorithms are the one in which only one process holding a special message called the token, may enter the critical section.

Suzuki-Kasami's algorithm: A process holding the token is allowed to enter into the critical section. A single process has the privilege and a node requesting critical section broadcasts a request message to all the other nodes. A process sends the privilege if the token is idle with the site. The site having a token can continuously enter the critical section until it sends the token to some other site. The request message has the format request (j, h j), which means site j is requesting its critical section. Each node
maintains an array RN of size N for recording the latest sequence number received from each of the other nodes. The TOKEN message has the format TOKEN (LN), where LN is an array of size N where LN[j] is the latest critical section executed by a node j. if RN[j] = LN[j]+1, it means that a node j has sent a request for its new sequence of critical section and the node having the privilege adds this to the queue and if token is idle, the node sends the TOKEN (LN) to the node requesting critical section. The number of messages per critical section entry is (N-1) REQUEST messages plus one TOKEN message so N messages in all or 0 if the node having the token wants to enter the critical section.

- When done with the critical section, process P i sets LNi[i] = RNi[i]
- For every process P j it appends P j in waiting queue if RNi[j] = LNi[j]+1
- If the waiting queue is not empty, it extracts the process at the head of the waiting queue and sends the token to that process

Suzuki-Kasami's algorithm based on causal ordering: Concurrent requests: Let Ri and Rj are two vectors of two processes Pi and Pj respectively. Definition: For any two time vectors R i and R j :

Ri ≤ Rj iff Ri ≤ Rj and it exists k such as Ri [k] < Rj[k]
Ri < Rj iff Ri ≤ Rj and it exists k such as Ri [k] < Rj[k]
Ri || Rj iff ¬ (Ri < Rj) and ¬ (Rj < Ri)

Principle: To implement the causal ordering, we use, for every process Pi the vector timestamp Ri where Ri [k] is the last request time sent by process Pk and received by Pi. The new requests received by process Pi are stored in a waiting local queue Qi.

When a process Pi holding the token, requests the critical section, it enters its critical section without sending the message. In another way, it increases Ri[i] by one, appends (i, Ri [i]) to Qi, sends the request “REQ (Qi)” to all other processes, sets Qi to empty and waits for the token.

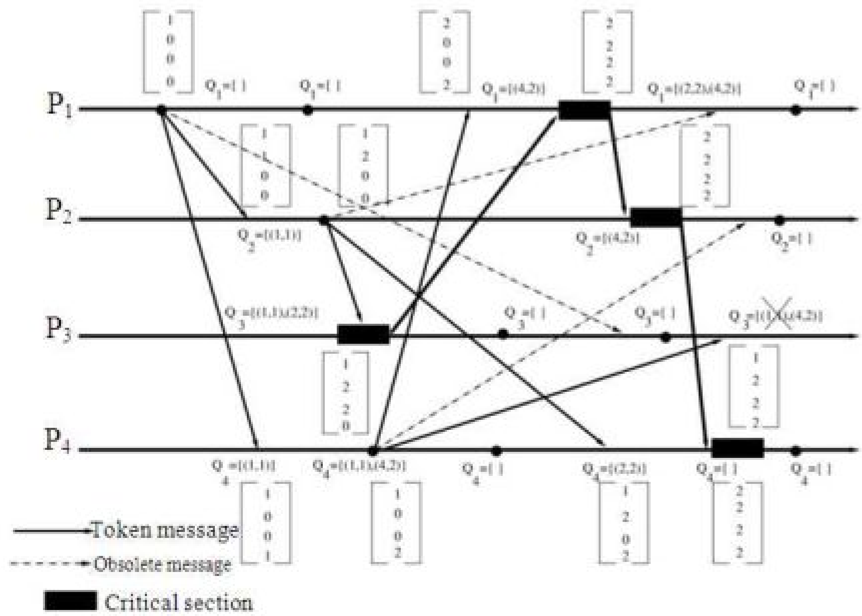


Fig: Mutual exclusion with causal ordering

When a process Pj receives a request “REQ (Q)” from another process, Pi removes from all queues Qi and Q the obsolete request and appends Q to Qi to obtain by merging a queue Qi . A process Pi holding the idle token, sends it to the head of its waiting local queue Qi and sets Qi to empty.

Local variable at process P:

R_i: Vector of timestamps where $R_i[i]$ denotes the last timestamp of requesting critical section by process $P_{i.}$

T: Vector of timestamps where T[i] denotes the last timestamp critical section execution by process P_i.

Q_i: Waiting Fifo queue of (j, $h_{j)}$ where j is the process P_j and h_j is the timestamp request.

HT_i: Boolean true if process P_i holds the token, false otherwise. Initially one process holds the token.

$InCS_i$: Boolean true if process P_i is in the critical section and false otherwise.

$Next_i$: Pointer denotes the next process to which, the token will be sent.

Messages of the algorithm: We consider two kinds of messages exchanged between processes:
REQ (Q): This message is sent to all others process to obtain the token.
TOKEN (Q, T): This message to denote the permission to enter the critical section.

```
// Message Structure
struct message{
   char *type;
   int  req;
   int  NODEID;
   int  *queue;
   int  lenqueue;
   int  *last;
   int  lenlast;
};
```

Algorithm: We define the concatenation operator “*” as follows: the operator “*” merges the waiting received Q and local Q i and we denote it by “Q*Q i ”. We consider the two following cases:

- When a process Pi receives a waiting queue Q attached to a token message, it deletes from Qi all obsolete messages. For all (k, h) є Q such than (k, h’) є Qi , remove (k, h) from Qi
- When a process Pi receives waiting queue Q attached to request message, it deletes from Q and Qi all obsolete messages

Rule₁: P_i requests the critical section

```
If (HT_i=False) Then
     R_i[i] ← R_i[i] +1
     Q_i ← Q_i*(i, R_i[i])
     For all k Send REQ (Q_i) To P_k
     Q_i ← [ ]
EndIf
```

Rule₂: P_i receives REQ (Q)

```
Q_i ← Q_i*Q
For all k є Q_i R_i[k] ← max (R_i[k], R[k])
R_i[i] ← max (R_i[k])
```

Rule₃: P_i receives TOKEN (Q, T)

```
HT_i ← True
For all k R_i[k] ← max (R_i[k], T[k])
Q_i ← Q_i*Q
InCS_i ← True
```

Rule₄: P_i releases the critical section

```
InCS_i ← False
T[i] ← R_i[i]
Next_i ← Head (Q_i)
If (Next_i ≠ Nil) Then
     HT_i ← False
     Q_i ← Remove (Head (Q_i))
     Send TOKEN (Q_i, T) To Next_i
     Next_i ← Nil
     Q_i ← [ ]
EndIf
```

Working of Part 2

- Initially, node 0 has the token.
- Every other node (which wants to enter CS) broadcast the ***IsREQ*** message to node 0 via TCP.
- Requests reach the node 0 in arbitrary order. The requests are stored in Queue in causal fashion i.e if node x sends the IsREQ msg first and it gets delayed(which is simulated by random sleep function) but still it will be at top of the queue.
- After node 0, releases the token. It sends back the Token to the node which is at the top of the queue. Message sent in this case is of the form ***IsTOKEN*** .
- Now, the node which receives the token will follow the same steps and this process goes on until there are no further requests for CS.

Screenshots:

1. **Node 0:-**

```
gunno@gunno:~/Documents/Sem_IV_IIITH/DS/Project/Mutual Exclusion$ ./MutualExclusion -p 8080
=========================================================================
Types Of Messages-
1. IsToken Message: Current node sends the token to next requesting node
2. IsReq Message: Current node sends request to enter in critical section
=========================================================================
Started Node: 0 on Port: 8080

Listening on PORT: 8080
==================================================
Node: 0 is entering into critical section.
Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [1] >
Node: 0 has exited the critical section.
==================================================

Node: 0 is sending a Released_Token_Message to Node: 1.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [1,0,0,0], From NodeID: [0]>

Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [1] >
Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [1] >

Node- 0 is sending a Request_To_Enter_CS_Message to Node- 1

Node- 0 is sending a Request_To_Enter_CS_Message to Node- 2

Node- 0 is sending a Request_To_Enter_CS_Message to Node- 3
Message received- < MsgType: [ISTOKEN], From NodeID: [1], Token Count: [1] >
-----------------------------------------
Node: 0 received the token from Node: 3
-----------------------------------------
==================================================
Node: 0 is entering into critical section.
Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [2] >
Node: 0 has exited the critical section.
==================================================

Node: 0 is sending a Released_Token_Message to Node: 1.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [2,1,1,1], From NodeID: [0]>

----------------------------------------------------
Completed
----------------------------------------------------
Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [2] >
Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [2] >
```

2. Node 1:-

```
=========================================================================
Types Of Messages-
1. IsToken Message: Current node sends the token to next requesting node
2. IsReq Message: Current node sends request to enter in critical section
=========================================================================
Started Node: 1 on Port: 8081

Listening on PORT: 8081

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 2

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 3
Message received- < MsgType: [ISTOKEN], From NodeID: [0], Token Count: [0] >
-----------------------------------------
Node: 1 received the token from Node: 0
-----------------------------------------
================================================
Node: 1 is entering into critical section.
Node: 1 has exited the critical section.
================================================
Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [1] >

Node: 1 is sending a Released_Token_Message to Node: 2.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [1,1,0,0], From NodeID: [1]>

Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [1] >
Message received- < MsgType: [ISREQ], From NodeID: [0], Request Count: [2] >

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 2

Node- 1 is sending a Request_To_Enter_CS_Message to Node- 3
Message received- < MsgType: [ISTOKEN], From NodeID: [1], Token Count: [1] >
-----------------------------------------
Node: 1 received the token from Node: 0
-----------------------------------------
================================================
Node: 1 is entering into critical section.
Node: 1 has exited the critical section.
================================================
Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [2] >

Node: 1 is sending a Released_Token_Message to Node: 2.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [2,2,1,1], From NodeID: [1]>

----------------------------------------------------
Completed
----------------------------------------------------
Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [2] >
```

3. Node 2:-

```
=========================================================================
Types Of Messages-
1. IsToken Message: Current node sends the token to next requesting node
2. IsReq Message: Current node sends request to enter in critical section
=========================================================================
Started Node: 2 on Port: 8082

Listening on PORT: 8082
Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [1] >

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 1

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 3
Message received- < MsgType: [ISTOKEN], From NodeID: [0], Token Count: [0] >
-----------------------------------------
Node: 2 received the token from Node: 1
-----------------------------------------
================================================
Node: 2 is entering into critical section.
Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [1] >
Node: 2 has exited the critical section.
================================================

Node: 2 is sending a Released_Token_Message to Node: 3.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [1,1,1,0], From NodeID: [2]>

Message received- < MsgType: [ISREQ], From NodeID: [0], Request Count: [2] >
Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [2] >

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 1

Node- 2 is sending a Request_To_Enter_CS_Message to Node- 3
Message received- < MsgType: [ISTOKEN], From NodeID: [1], Token Count: [1] >
-----------------------------------------
Node: 2 received the token from Node: 1
-----------------------------------------
================================================
Node: 2 is entering into critical section.
Node: 2 has exited the critical section.
================================================
Message received- < MsgType: [ISREQ], From NodeID: [3], Request Count: [2] >

Node: 2 is sending a Released_Token_Message to Node: 3.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [2,2,2,1], From NodeID: [2]>

----------------------------------------------------
Completed
----------------------------------------------------
```

4. Node 3:-

```
gunno@gunno:~/Documents/Sem_IV_IIITH/DS/Project/Mutual Exclusion$ ./MutualExclusion -p 8083
=========================================================================
Types Of Messages-
1. IsToken Message: Current node sends the token to next requesting node
2. IsReq Message: Current node sends request to enter in critical section
=========================================================================
Started Node: 3 on Port: 8083

Listening on PORT: 8083
Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [1] >
Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [1] >

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 1

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 2
Message received- < MsgType: [ISTOKEN], From NodeID: [1], Token Count: [0] >
-----------------------------------------
Node: 3 received the token from Node: 2
-----------------------------------------
================================================
Node: 3 is entering into critical section.
Message received- < MsgType: [ISREQ], From NodeID: [0], Request Count: [2] >
Node: 3 has exited the critical section.
================================================

Node: 3 is sending a Released_Token_Message to Node: 0.
Message: <QueueLength: [0], QueueData: [], TotalNodes: [4], TokenCount Array: [1,1,1,1], From NodeID: [3]>

Message received- < MsgType: [ISREQ], From NodeID: [1], Request Count: [2] >
Message received- < MsgType: [ISREQ], From NodeID: [2], Request Count: [2] >

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 0

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 1

Node- 3 is sending a Request_To_Enter_CS_Message to Node- 2
Message received- < MsgType: [ISTOKEN], From NodeID: [2], Token Count: [1] >
-----------------------------------------
Node: 3 received the token from Node: 2
-----------------------------------------
================================================
Node: 3 is entering into critical section.
Node: 3 has exited the critical section.
================================================
```

Execution of part 2

- ***Compilation***
  gcc MutualExclusion.c -lm -lpthread -o MutualExclusion
  (OR)
  make

- ***Running different nodes***
  ./MutualExclusion -p <port_number_node>

  Node 0
  ./MutualExclusion -p 8080

  Node 1
  ./MutualExclusion -p 8081

  Node 2
  ./MutualExclusion -p 8082

  Node 3
  ./MutualExclusion -p 8083

REFERENCES -

1. **“Efficient group Mutual Exclusion protocols for Message Passing Distributed Computing Systems” by Abhishek Swaroop**
2. **“Distributed Mutual Exclusion Based on Causal Ordering” by Naimi and Thiare**
3. **https://www.cs.miami.edu/home/burt/learning/Csc609.051/notes/02.html**
4. **https://www.includehelp.com/cryptography/vernam-cipher.aspx**
5. **http://www.cs.fsu.edu/~xyuan/cop5611/lecture5.html**