Distributed Systems Project

Simulate message delivery guarantees such as FIFO and Arbitrary, and their impact on some Mutual Exclusion Distributed Algorithm

Ankit Pant 2018201035
Harshita Agrawal 2018201014
Ravi Jakhania 2018201018

Submitted on: April 23, 2020

Contents

1 Introduction

Distributed systems are quite ubiquitous these days owing to the large amounts of data that is being generated and processed. It is essential that when the computing infrastructure is distributed, the communication between these components is robust. Though the advancement in hardware and network technologies have made large scale distributed systems possible, we still need a layer of dependable software systems that makes reliable communication possible between the distributed system and help manage the various components. Since various systems are communicating with each other and may be accessing some common resources, it is also imperative that we have some algorithms in place that prevent more than one system from accessing the critical resource at a time and prevent race conditions. We do this by using Distributed Mutual Exclusion Algorithms.

This project aims to explore and simulate two modes or channels of communication i.e. First In First Out or *FIFO* channels and *Arbitrary* channels. A more detailed description of them are presented in the following sections. After the implementation of the *FIFO* and *Arbitrary* channels, we simulate a Distributed Mutual Exclusion Algorithm known as the Lamport's Mutual Exclusion Algorithm on the channels and measure the impact of the communication channels on the distributed mutual exclusion algorithm.

The following sections elaborate more on the literature as well as describe the methodology as well as the experimental setup and results obtained. The report ends by reporting on the conclusions and suggesting improvements as future scope.

2 Literature Review

This section theoretically elaborates on the various terms and components of the project. This section also delves into the components of the distributed systems that have been used to complete the project. Each such component is elaborated in the following subsections:

2.1 Message Ordering and Group Communication

The importance of group communication cannot be over-stated in distributed systems. In most scenarios, a node is able to communicate with all other nodes that are part of the distributed system. Apart from the aspect of nodes being able to communicate with each other, the order in which the messages are delivered is also important as it determines the order of execution of commands and also helps to keep the system consistent. There are several such message orderings, a few of which are elaborated in the following subsections.

2.1.1 Arbitrary Order

Arbitrary order implies that there is no ordering between messages sent by one node to the other. This message ordering is also called *non-FIFO* ordering. For

example if a process 'P1' sends two messages (m1 and m2) to 'P2' and the timestamp of message *m1* is *t1* and for message *m2* is *t2* where $t1 < t2$ the process 'P2' may either receive message *m1* before message *m2* or it may receive message *m2* before message *m1*.

2.1.2 FIFO Order

FIFO order or First In First Out ordering implies that if messages are sent by a process to another process in a particular order, they are received by the other process in the same order that they were sent [1]. For instance if a process 'P1' sends two messages (m1 and m2) to 'P2' and the timestamp of message *m1* is *t1* and for message *m2* is *t2* where $t1 < t2$ the process 'P2' must necessarily receive message *m1* before message *m2*. In this message ordering it is ensured that multicast from each sender are received in the order they are sent, at all receivers. The relative ordering between multicast messages from different senders is not important.

2.1.3 Causal Ordering

In causal ordering implies that if two send (or more) events happen then their corresponding receive events happen in the same order as well throughout the distributed system. It is a stricter version of *FIFO* ordering and the relative ordering between multicast messages from different senders is also taken into account. Thus for example if there are two messages *m1* and *m2* then if send timestamp(*m1*) $<$ send timestamp(*m2*), then for each destination of those messages, receive timestamp(*m1*) $<$ receive timestamp(*m2*) [2].

2.2 Distributed Mutual Exclusion Algorithms

Mutual exclusion is one of the core problems when running distributed systems. It needs to be ensured that more than two nodes are not modifying the same object and not creating race conditions. Thus, mutual exclusion states that only one process must be executing its critical section at a given point in time. Since there is no traditional notion of shared memory in distributed systems, the solutions like semaphores cannot be used. Hence it results in a need to implement distributed mutual exclusion algorithms. Distributed mutual exclusion algorithms are implemented using these basic approaches [3]:

- Token based approach
- Non-token based approach
- Quorum based approach

The following sub sub-section briefly discusses a token based distributed mutual exclusion algorithm which has been simulated in this project.

2.2.1 Lamport's algorithm

Lamport's algorithm is a Token based Distributed Mutual Exclusion algorithm. This algorithm executes the critical section requests from various processes in the increasing order of timestamps. Every node is also required to keep a queue called the *request queue* which stores the critical section requests in the increasing order of timestamps. However it is required that the channel delivers messages in *FIFO* order for this algorithm to execute correctly. The algorithm has three phases [4] namely:

- Requesting the critical section
- Executing the critical section
- Releasing the critical section

The following diagram 2.2.1 provides the pseudocode for the algorithm.

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of $request\_queue_i$.

Releasing the critical section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

Figure 1: Pseudocode for Lamport's Algorithm [4]

3 Methodology

This sections describes the methodology of the project. First and foremost, two message ordering modes were created, namely *FIFO* and *Arbitrary*. After creating these message ordering modes, Lamport's Algorithm was run on top of the channels preserving the *FIFO* and *Arbitrary* message ordering respectively. Using

the Lamport's Algorithm, a dummy *critical section* of each node was run and it was ensured that only one node can execute its critical section at one time and the order of execution is based on the timestamp of request to execute critical with the lowest timestamp given the highest priority.

The performance of the algorithm was then examined by running it on both the message ordering modes and the impact of the message ordering mode was measured on the algorithm, primarily based on the correctness. The following sub-sections elaborate on the various aspects of the project.

3.1 System Model

To run the simulation we assume a distributed system with six nodes. It is also assumed that all nodes are non-Byzantine. Since the channel is reliable, message delivery guarantee is ensured and if a node communicates with any other node, it is guaranteed that the other node will receive the message. Each node is fully capable to send and receive messages as well as request (to execute) the critical section. Each node is also capable of communicating with every other node as this is also one of the pre-requisites of the Lamport's Algorithm. The implementation also handles arbitrary node (channel) failure by using a timeout mechanism.

3.2 Development Environment

The project was developed in python and a bash script was written to run the various nodes. Multithreading was used to manage asynchronous sends and receives. The distributed system so created was tested in bash terminal where each node used the IP address of *localhost* with different *port numbers*.

3.3 Creating FIFO Ordered Channel

The FIFO ordered channel was created using TCP connections for maintaining reliability and message delivery guarantee. Each node has a send and receive method (with helper functions) to be able to send messages to other nodes and receive messages from other nodes respectively.

3.4 Creating Arbitrary Ordered Channel

Arbitrary channels were simulated by adding a random delay while sending each message. This effectively simulated out of order delivery to the recipient node. This channel also has TCP as the underlying protocol to ensure reliability and message delivery guarantee.

3.5 Implementing Lamport's Algorithm

Lamport's algorithm was implemented such that it may run on each of the ordered channels. As required by the algorithm each node maintains a priority queue in which all critical section requests are stored in increasing order of timestamps.

Each node also maintains a local clock for timestamping purposes. The three types of messages, *Request*,*Reply*, and *Release* (which are vital to the algorithm) are also exchanged as required.

3.6 Testing

The testing phase consists of three sub-parts namely testing the FIFO ordered channel, testing the Arbitrary ordered channel, and testing mutual exclusion algorithm on both the ordered channels. A driver program is written that gives a menu-based interface to run either of the test cases. The test cases are written out in *csv* files that makes it easy to test variety of situations without having to hard-code the test cases.

4 Experimentation and Results

The test cases were run using the *Test.sh* bash script that starts all nodes in the Distributed System and gives a menu-based choice on which test to run. The inputs were taken from the *Check_channel.csv* file to test the *FIFO* ordered and *Arbitrary* ordered channels, and from the *ME_Test.csv* file to run the Lamport's mutual exclusion algorithm on both channels consecutively.

4.1 Testing the Message Ordering

The *FIFO* and *Arbitrary* Ordered channels created, were tested and were found to run correctly. Messages send from a process to another are received in the same order they were sent for the *FIFO* ordered channel. Messages sent by a process through the *Arbitrary* channel were received arbitrarily as expected. The results were written to two separate files, *FIFO_Result.txt* and *Arbitrary_Result.txt* respectively. Figure 2 shows the sample test file, while Figures 3 and Figure 4 show the output of the test file on *FIFO* ordered and *Arbitrary* ordered channels respectively. The test file has comma separated entries where in each row the first entry corresponds to sender node, the second entry corresponds to the receiver node and the third entry corresponds to the message.

Channel_test.csv

```
1  Node1,Node2,Hello_Node2_From_Node1_Msg#1
2  Node2,Node1,Hi_Node1_From_Node2_Msg#1
3  Node2,Node1,Hi_Again_Node1_From_Node2_Msg#2
4  Node2,Node1,Hi_Again_Node1_From_Node2_Msg#3
5  Node1,Node2,Hello_Node2_From_Node1_Msg#2
6  Node1,Node2,Hello_Node2_From_Node1_Msg#3
7  Node1,Node6,Hello_Node6_From_Node1_Msg#1
8  Node6,Node1,Hola_Node1_From_Node6_Msg#1
9  Node3,Node4,Aloha_Node4_From_Node3_Msg#1
10 Node3,Node5,Aloha_Node5_From_Node3_Msg#1
11 Node3,Node5,Aloha_Node5_From_Node3_Msg#2
12 Node3,Node4,Aloha_Node4_From_Node3_Msg#2
13
```

Figure 2: A sample test file

FIFO_Result.txt

```
1  Order of message processed
2
3  Node2: Received message: Hello_Node2_From_Node1_Msg#1 from address 127.0.0.1 : 59236
4
5  Node1: Received message: Hi_Node1_From_Node2_Msg#1 from address 127.0.0.1 : 53624
6
7  Node1: Received message: Hi_Again_Node1_From_Node2_Msg#2 from address 127.0.0.1 : 53628
8
9  Node1: Received message: Hi_Again_Node1_From_Node2_Msg#3 from address 127.0.0.1 : 53634
10
11 Node2: Received message: Hello_Node2_From_Node1_Msg#2 from address 127.0.0.1 : 59250
12
13 Node2: Received message: Hello_Node2_From_Node1_Msg#3 from address 127.0.0.1 : 59254
14
15 Node6: Received message: Hello_Node6_From_Node1_Msg#1 from address 127.0.0.1 : 33450
16
17 Node1: Received message: Hola_Node1_From_Node6_Msg#1 from address 127.0.0.1 : 53648
18
19 Node4: Received message: Aloha_Node4_From_Node3_Msg#1 from address 127.0.0.1 : 57610
20
21 Node5: Received message: Aloha_Node5_From_Node3_Msg#1 from address 127.0.0.1 : 49592
22
23 Node5: Received message: Aloha_Node5_From_Node3_Msg#2 from address 127.0.0.1 : 49596
24
25 Node4: Received message: Aloha_Node4_From_Node3_Msg#2 from address 127.0.0.1 : 57624
26
27
```

Figure 3: Output on FIFO ordered channel for the test file

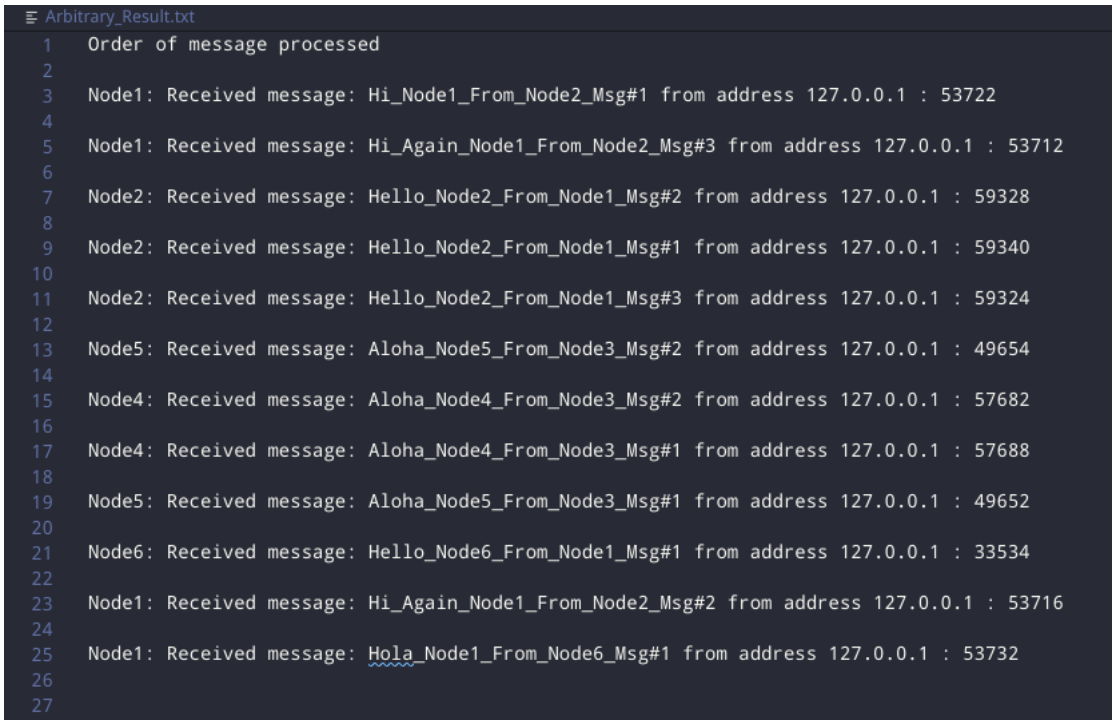

Figure 4: Output on Arbitrary ordered channel for the test file

Hence it can be seen from the figures above that the channels work as expected.

4.2 Testing Mutual Exclusion Algorithm

The Lamport's algorithm was then run on both *FIFO* ordered channel and *Arbitrary* ordered channel and found to give correct result for the *FIFO* ordered channel. For the *Arbitrary* ordered channel, Lamport's Algorithm fails as was expected and more than one process succeeds in securing the critical section since the messages are not delivered in FIFO order. The output of the algorithm was written to the file *Mutual_Execlusion_Result.txt*. Figure 5 shows the sample test file while the figure 6 and 7 show the output corresponding to running the algorithm in *FIFO* ordered channel and arbitrary ordered channel respectively. The test file stores the order in which the nodes should execute the critical section.

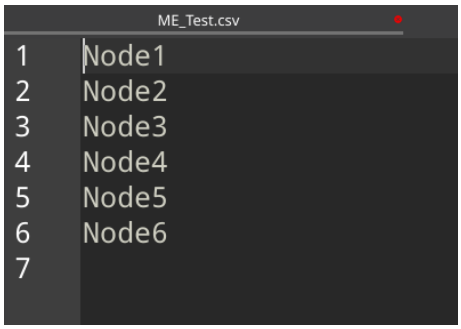

Figure 5: A sample test file

Mutual_Execlusion_Result.txt

```
Using FIFO order of channel
Processes Requests with Timestamp and process (Tsi,i)
(1, 'Node1')
(1, 'Node2')
(1, 'Node6')
(1, 'Node4')
(1, 'Node3')
(1, 'Node5')

Correct Order of processes to execute critical section is:
Node1
Node2
Node3
Node4
Node5
Node6

Actual Order of processes to execute critical section is:
Executing critical section of Node1
Exiting critical section of Node1
Executing critical section of Node2
Exiting critical section of Node2
Executing critical section of Node3
Exiting critical section of Node3
Executing critical section of Node4
Exiting critical section of Node4
Executing critical section of Node5
Exiting critical section of Node5
Executing critical section of Node6
Exiting critical section of Node6
```

Figure 6: Output on FIFO ordered channel for the test file

```
Using Arbitrary order of channel
Processes Requests with Timestamp and process (Tsi,i)
(1, 'Node1')
(1, 'Node2')
(1, 'Node3')
(1, 'Node4')
(1, 'Node5')
(1, 'Node6')

Correct Order of processes to execute critical section is:
Node1
Node2
Node3
Node4
Node5
Node6

Actual Order of processes to execute critical section is:
Executing critical section of Node2
Executing critical section of Node1
Exiting critical section of Node2
Exiting critical section of Node1
Executing critical section of Node3
Exiting critical section of Node3
Executing critical section of Node4
Exiting critical section of Node4
Executing critical section of Node5
Exiting critical section of Node5
Executing critical section of Node6
Exiting critical section of Node6
```

Figure 7: Output on Arbitrary ordered channel for the test file

Hence through the figures above it can be seen that when Lamport's Algorithm is run for *FIFO* ordered channel, it produces the correct output. However the algorithm fails when it is run on an *Arbitrary* Ordered Channel.

4.3 Impact of Channel on Mutual Exclusion Algorithm

This sub-section describes the impact of running the Distributed Mutual Exclusion Algorithm (Lamport's Algorithm) on *FIFO* ordered and *Arbitrary* ordered channels.

4.3.1 Correctness

Correctness of the algorithm running on the two channels is compared. Since Lamport's Algorithm necessarily requires a *FIFO* ordered channel for its correct execution, it does not run correctly on running in in an *Arbitary* ordered channel.

FIFO Ordered Channel: Algorithm runs correctly.

Arbitrary Ordered Channel: Algorithm fails.

4.3.2 Number of Messages

Number of messages exchanged when running on the two channels is compared. Since execution is done on reliable channels (using TCP), we see no difference in the number of messages exchanged.

FIFO Ordered Channel: $3(N-1)$.

Arbitrary Ordered Channel: $3(N-1)$.

4.3.3 Algorithmic Runtime

Since the communication between nodes in the distribution system can have uncertain delays (in spite of guaranteed message delivery), the total runtime of executing Lamport's algorithm on either channel was not measured as it was not well understood how to eliminate the effect of random network delays while running the algorithm.

5 Conclusion

It is necessary for a distributed system to have mutual exclusion for its proper functioning. The ordering of messages in channels connecting the nodes also play a critical part in supporting the algorithms that can be run on the system. Through this project, two such message ordering channels, namely *FIFO* ordered and *Arbitrary* ordered channels were explored. The channels were simulated and after that a distributed mutual exclusion algorithm (Lamport's algorithm) was run on them to measure the impact of message ordering on the algorithm.

It was found that the ordering in channels have a significant impact on Lamport's Mutual Exclusion algorithm and while *FIFO* ordered channels runs the algorithm without a problem, *Arbitrary* ordered channel can cause the algorithm to fail. Hence through this project, we conclude that Lamport's Mutual Exclusion algorithm requires (at least) a *FIFO* ordered channel and it may fail if we cannot guarantee the First In First Out message ordering.

6 Future Scope

Currently Lamport's algorithm fails for *Arbitrary* ordered channels. The modifications can be made to have Lamport's algorithm run on *Arbitrary* ordered channels too. Resolution may be done by:

- Adding buffer at receivers
- Buffer orders messages correctly by timestamp
- After all messages are received and correctly ordered, forward them to the mutual exclusion algorithm

References

[1] *FIFO executions*, Ajay D. Kshemkalyani & Mukesh Singhal, Page 191, Distributed Computing Principles, Algorithms, and Systems.

[2] *Causally ordered (CO) executions*, Ajay D. Kshemkalyani & Mukesh Singhal, Page 191, Distributed Computing Principles, Algorithms, and Systems.

[3] *Distributed mutual exclusion algorithms*, Ajay D. Kshemkalyani & Mukesh Singhal, Page 305, Distributed Computing Principles, Algorithms, and Systems.

[4] *Lamport's Algorithm*, Ajay D. Kshemkalyani & Mukesh Singhal, Page 309, Distributed Computing Principles, Algorithms, and Systems.