LAB NO: 6&7

CLOCK SYNCHRONIZATION & MUTUAL EXCLUSION (ELECTION ALGORITHM)

I. Introduction to Clock Synchronization

Time notion: Each computer is equipped with a physical (hardware) clock.
At time t, the operating system (OS) of a process i reads the hardware clock H (t) of the processor, generates the software clock C=a H(t)+b, as a counter incremented by ticks of an oscillator.

Time in Distributed System (DS): Time is a key factor in a DS to analyse how distributed execution evolve.

Problems: Lacking of a global reference time: it's hard to know the state of a process during a distributed computation. However, it's important for processes to share a common time notion.

The techniques used to coordinate a common time notion among processes is known as Clock Synchronization.

Clock Synchronization: The hardware clock of a set of computers may differ because they count time with different frequencies.

Clock Synchronization faces this problem by means of synchronization algorithms

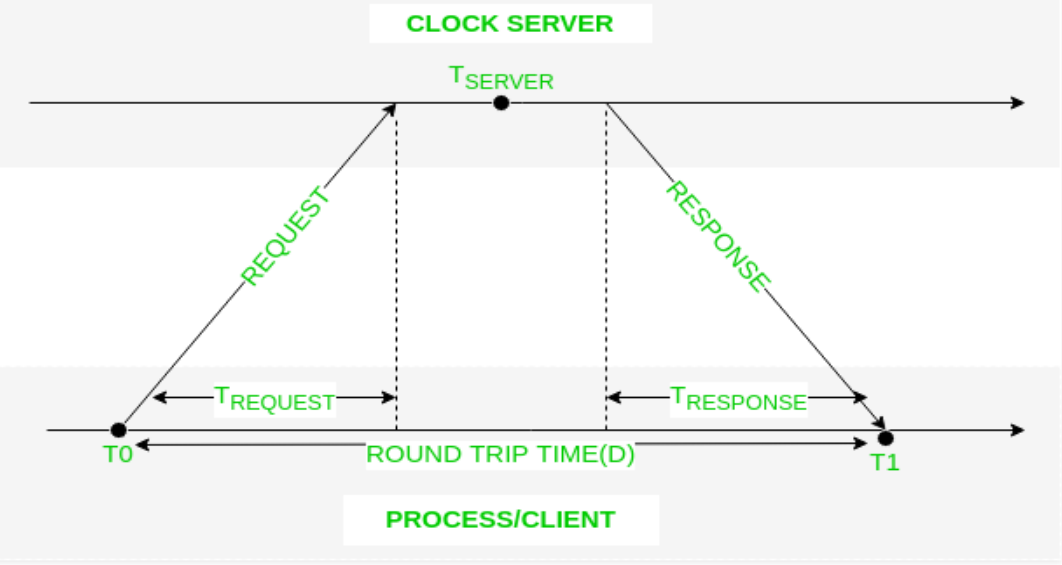
- Standard communication infrastructure
- No additional hardware

Clock synchronization algorithms

- External
- Internal
- Heartbeat (pulse)

External clock synchronization:
Is the one in which an external reference clock is present. It is used as a reference and the nodes in the system can set and adjust their time accordingly.
Internal clock synchronization: Is the one in which each node shares its time with other nodes and all the nodes set and adjust their times accordingly.

II. **Cristian's algorithm: Cristian's Algorithm** is a clock synchronization algorithm is used to synchronize time with a time server by client processes. This algorithm works well with low-latency networks where Round Trip Time is short as compared to accuracy while redundancy prone distributed systems/applications do not go hand in hand in this algorithm. Here Round Trip Time refers to the time duration between start of a Request and end of corresponding Response.



Algorithm:

1) The process on the client machine sends the request for fetching clock time (time at server) to the Clock Server at time.

2) The Clock Server listens to the request made by the client process and returns the response in form of clock server time.

3) The client process fetches the response from the Clock Server at time and calculates the synchronised client clock time using the formula given below.

$T_{CLIENT} = T_{SERVER} + (T1-T0)/2$

T_{CLIENT} = synchronised clock time

T_{SERVER} = clock time returned by the server

T0 = time at which request was sent by the client process

T1= time at which response was received by the client process

III. **Berkeley's algorithm**: Berkeley's Algorithm is a clock synchronization technique used in distributed systems. The algorithm assumes that each machine node in the network either doesn't have an accurate time source or doesn't possess an UTC server.

Algorithm:

1) An individual node is chosen as the master node from a pool nodes in the network. This node

is the main node in the network which acts as a master and rest of the nodes act as slaves. Master node is chosen using a election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

3) Master node calculates average time difference between all the clock times received and the clock time given by master's system clock itself. This average time difference is added to the current time at master's system clock and broadcasted over the network.
Pseudocode for above step:

```
# receiving time from all slave nodes

repeat_for_all_slaves:

 time_at_slave_node = receive_time_at_slave()

 # calculating time difference

 time_difference = time_at_master_node - time_at_slave_node

# average time difference calculation

average_time_difference = sum(all_time_differences) / number_of_slaves

synchronized_time = current_master_time + average_time_difference

# broadcasting synchronized to whole network

broadcast_time_to_all_slaves(synchronized_time)
```

Diagram below illustrates how the master sends request to slave nodes.

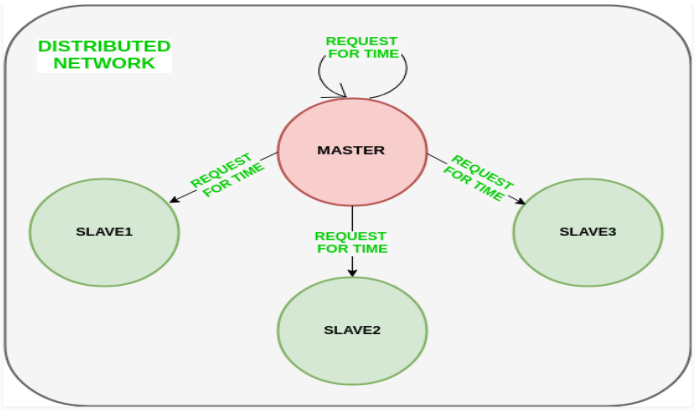


Diagram below illustrates how slave nodes send back time given by their system clock.

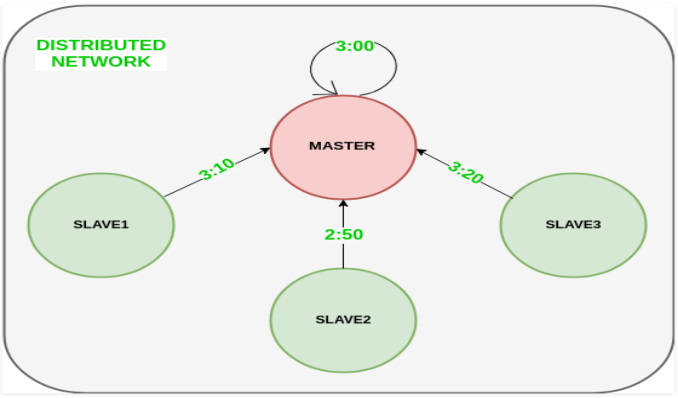
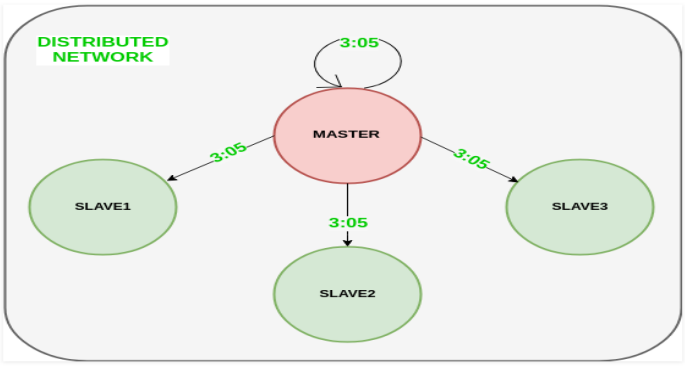


Diagram below illustrates the last step of Berkeley's algorithm.



IV. Solved examples:

A. Cristian's algorithm

I) **To initiate a prototype of a clock server on local machine:**

Server:

Python3 program imitating a clock server

```
import socket
import datetime
import time
# function used to initiate the Clock Server
def initiateClockServer():
        s = socket.socket()
        print("Socket successfully created")
        # Server port
        port = 8011
        s.bind(('', port))
        # Start listening to requests
        s.listen(5)
        print("Socket is listening...")
        # Clock Server Running forever
        while True:
        # Establish connection with client
                connection, address = s.accept()
                print('Server connected to', address)
        # Respond the client with server clock time
                connection.send(str(datetime.datetime.now()).encode())
        # Close the connection with the client process
        connection.close()
# Driver function
if __name__ == '__main__':
        # Trigger the Clock Server
        initiateClockServer()
```

Output Sever Side:

```
Ubuntu@ubuntu:~$ python3 cris_ser.py
Socket successfully created
```

Socket is listening...

Server connected to ('127.0.0.1', 42310)

II) **Code below is used to initiate a prototype of a client process on local machine:**

```
# Python3 program imitating a client process

import socket
import datetime
from dateutil import parser
from timeit import default_timer as timer


# function used to Synchronize client process time
def synchronizeTime():

    s = socket.socket()

    # Server port
    port = 8011

    # connect to the clock server on local computer
    s.connect(('127.0.0.1', port))

    request_time = timer()

    # receive data from the server
    server_time = parser.parse(s.recv(1024).decode())
    response_time = timer()
    actual_time = datetime.datetime.now()
```

```
    print("Time returned by server: " + str(server_time))

    process_delay_latency = response_time - request_time

    print("Process Delay latency: " + str(process_delay_latency) + " seconds")

    print("Actual clock time at client side: " + str(actual_time))

    # synchronize process client clock time
    client_time = server_time + datetime.timedelta(seconds = (process_delay_latency) / 2)

    print("Synchronized process client time: " + str(client_time))

    # calculate synchronization error
    error = actual_time - client_time
    print("Synchronization error : " + str(error.total_seconds()) + " seconds")

    s.close()



# Driver function
if __name__ == '__main__':

    # synchronize time using clock server
    synchronizeTime()
```

Output Client Side:

Time returned by server: 2021-05-27 17:41:36.223936

Process Delay latency: 0.000858066999995799 seconds

Actual clock time at client side: 2021-05-27 17:41:36.224587

Synchronized process client time: 2021-05-27 17:41:36.224365

Synchronization error : 0.000222 second

B. Berkeley's algorithm:

Server Side:

 # Python3 program imitating a clock server

```
from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time


# datastructure used to store client address and clock data
client_data = {}


''' nested thread function used to receive
    clock time from a connected client '''
def startRecieveingClockTime(connector, address):

    while True:
        # recieve clock time
```

```
        clock_time_string = connector.recv(1024).decode()
        clock_time = parser.parse(clock_time_string)
        clock_time_diff = datetime.datetime.now() - \
                                        clock_time

        client_data[address] = {
                    "clock_time"      : clock_time,
                    "time_difference" : clock_time_diff,
                    "connector"       : connector
                    }

        print("Client Data updated with: "+ str(address),
                                        end = "\n\n")
        time.sleep(5)


''' master thread function used to open portal for
    accepting clients over given port '''
def startConnecting(master_server):

    # fetch clock time at slaves / clients
    while True:
        # accepting a client / slave clock client
        master_slave_connector, addr = master_server.accept()
        slave_address = str(addr[0]) + ":" + str(addr[1])

        print(slave_address + " got connected successfully")

        current_thread = threading.Thread(
                    target = startRecieveingClockTime,
```

```
                    args = (master_slave_connector,
                                    slave_address, ))
        current_thread.start()


# subroutine function used to fetch average clock difference
def getAverageClockDiff():

    current_client_data = client_data.copy()

    time_difference_list = list(client['time_difference']
                                for client_addr, client
                                   in client_data.items())


    sum_of_clock_difference = sum(time_difference_list, \
                                datetime.timedelta(0, 0))

    average_clock_difference = sum_of_clock_difference \
                                    / len(client_data)

    return  average_clock_difference

def synchronizeAllClocks():

    while True:

        print("New synchroniztion cycle started.")
        print("Number of clients to be synchronized: " + \
                                    str(len(client_data)))
```

```
        if len(client_data) > 0:

            average_clock_difference = getAverageClockDiff()

            for client_addr, client in client_data.items():
                try:
                    synchronized_time = \
                        datetime.datetime.now() + \
                                average_clock_difference

                    client['connector'].send(str(
                            synchronized_time).encode())

                except Exception as e:
                    print("Something went wrong while " + \
                        "sending synchronized time " + \
                        "through " + str(client_addr))

        else :
            print("No client data." + \
                    " Synchronization not applicable.")

        print("\n\n")

        time.sleep(5)


# function used to initiate the Clock Server / Master Node
def initiateClockServer(port = 8080):
```

```
    master_server = socket.socket()
    master_server.setsockopt(socket.SOL_SOCKET,
                             socket.SO_REUSEADDR, 1)

    print("Socket at master node created successfully\n")

    master_server.bind(('', port))

    # Start listening to requests
    master_server.listen(10)
    print("Clock server started...\n")

    # start making connections
    print("Starting to make connections...\n")
    master_thread = threading.Thread(
                    target = startConnecting,
                    args = (master_server, ))
    master_thread.start()

    # start synchroniztion
    print("Starting synchronization parallely...\n")
    sync_thread = threading.Thread(
                      target = synchronizeAllClocks,
                      args = ())
    sync_thread.start()



# Driver function
```

```
if __name__ == '__main__':

    # Trigger the Clock Server
    initiateClockServer(port = 8080)
```

Output:

```
New synchroniztion cycle started.
Number of clients to be synchronized: 2


Client Data updated with: 127.0.0.1:37624

Client Data updated with: 127.0.0.1:37626
```

Client Side :

 # Python3 program imitating a client process

```
from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time


# client thread function used to send time at client side
def startSendingTime(slave_client):
```

```
    while True:
        # provide server with clock time at the client
        slave_client.send(str(
                    datetime.datetime.now()).encode())

        print("Recent time sent successfully",
                                    end = "\n\n")
        time.sleep(5)


# client thread function used to receive synchronized time
def startReceivingTime(slave_client):

    while True:
        # receive data from the server
        Synchronized_time = parser.parse(
                        slave_client.recv(1024).decode())

        print("Synchronized time at the client is: " + \
                                str(Synchronized_time),
                                end = "\n\n")


# function used to Synchronize client process time
def initiateSlaveClient(port = 8080):

    slave_client = socket.socket()

    # connect to the clock server on local computer
    slave_client.connect(('127.0.0.1', port))
```

```
    # start sending time to server
    print("Starting to receive time from server\n")
    send_time_thread = threading.Thread(
                    target = startSendingTime,
                    args = (slave_client, ))
    send_time_thread.start()


    # start recieving synchronized from server
    print("Starting to recieving " + \
                      "synchronized time from server\n")
    receive_time_thread = threading.Thread(
                     target = startReceivingTime,
                     args = (slave_client, ))
    receive_time_thread.start()


# Driver function
if __name__ == '__main__':

    # initialize the Slave / Client
    initiateSlaveClient(port = 8080)
```

Output:

Synchronized time at the client is: 2021-05-27 17:45:29.885369

Recent time sent successfully

Synchronized time at the client is: 2021-05-27 17:45:34.888644

Recent time sent successfully

V. ELECTION ALGORITHMS

Theory:

Many algorithms used in the distributed system require a coordinator that performs functions needed by other processes in the system. Election algorithms are designed to choose a coordinator.

Election Algorithms

Election algorithms choose a process from a group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on another processor. E.g. coordinator process is picked as a master in Berkeley clock synchronization algorithm. Election algorithm determines, where a new copy of a coordinator should be restarted. Election algorithm assumes that every active process in the system has a unique priority number. The process with the highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has the highest priority number. The active highest selected number is then sent to every active process in the distributed system.

We have two election algorithms for two different configurations of the distributed system.

1. The Bully Algorithm

This algorithm applies to a system where every process can send a message to every other process in the system.

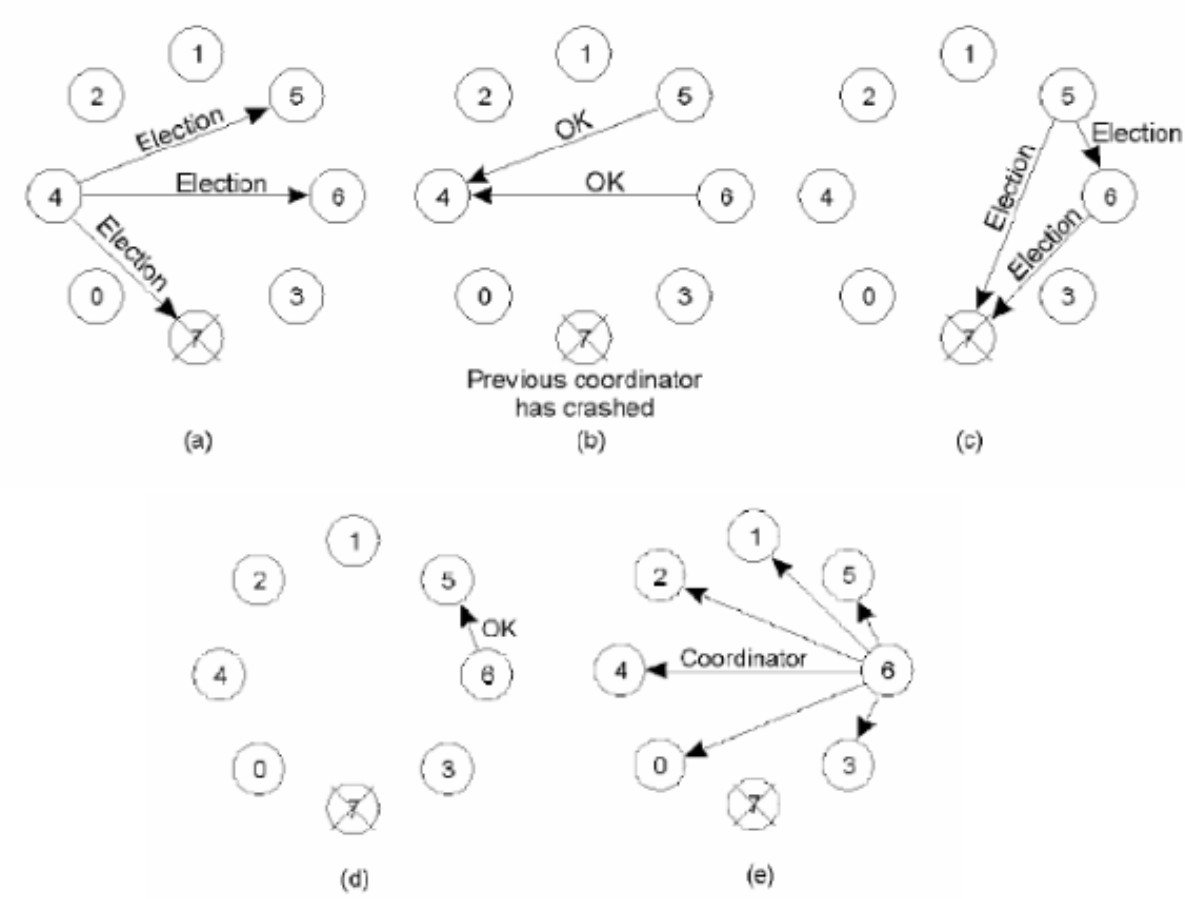
Assumptions

- Each process knows the ID and address of every other process
- Communication is reliable
- A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
- Three types of messages: Election, OK, Coordinator
- Several processes can initiate an election simultaneously
- Need consistent result

Algorithm – Suppose process P sends a message to the coordinator node.

1. If the coordinator node does not respond to it within a time interval T, then it is assumed that the coordinator node has failed.
2. P sends Election messages to all process with higher IDs and awaits OK messages

3. If no OK messages, P becomes coordinator and sends Coordinator messages to all processes with lower IDs
4. If it receives an OK, it drops out and waits for a Coordinator message
5. If a process receives an Election message
   Immediately sends Coordinator message if it is the process with the highest ID
   Otherwise,
      Returns an OK and starts an election
6. If a process receives a Coordinator message, it treats a sender as the coordinator



Bully Algorithm Example

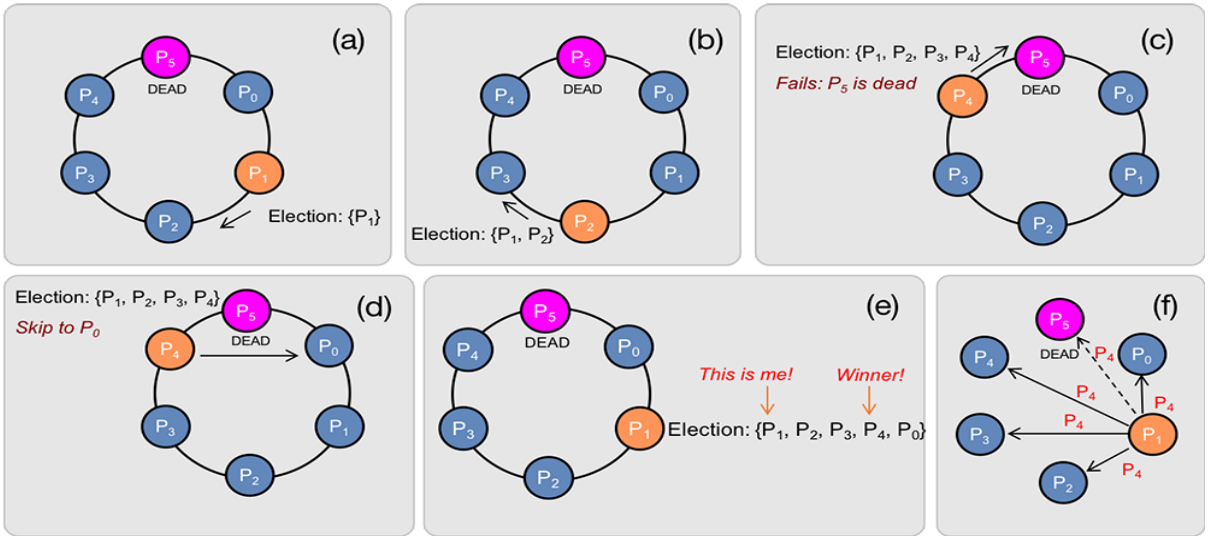
2. The Ring Algorithm

This algorithm applies to systems organized as a ring (logically or physically). In this algorithm, we assume that the link between the processes are unidirectional, and every process can message to the process on its right only.

Assumes that processes are logically ordered in some fashion, and that each process knows the order and who is the coordinator. No token is involved. When a process notices that the coordinator is not responding it sends an ELECTION message with its own id to its downstream neighbour. If that neighbour doesn't acknowledge, it sends it to its neighbour's neighbour, etc. Each station that receives the ELECTION message adds its own id to the list. When the message circulates back to the originator, it selects the highest id in the list and sends a COORDINATOR message announcing the new coordinator. This message circulates once and is removed by the originator. If two elections are held simultaneously (say because two different processes notice simultaneously that the coordinator is dead) then each comes up with the same list and elects the same coordinator. Some time is wasted, but this really hurts nothing.

Algorithm:-

1. Processes are arranged in a logical ring; each process knows the structure of the ring
2. A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
3. The initiator sends Election message to the closest downstream node that is alive
   – Election message is forwarded around the ring
   – Each process adds its own ID to the Election message
4. When Election message comes back, initiator picks node with highest ID and sends a Coordinator message specifying the winner of the election
   – Coordinator message is removed when it has circulated once.



. Ring Algorithm Example

VI. Lab Exercises:

1. The Manipal Foodie is a renowned automated food processing outlet known for its tiffin service to students. The various processes involved are food production, filling and packing. Every day more than 3000 orders are received on an average from the students in manipal. There are total of 4 production lines for orders received from KMC, MIT, TAPMI and SOLS students, each of them has a digital clock which needs to be in synchronization with the master clock. The master clock mounted in the testing lab controls the entire clock system. Design an appropriate solution using Berkeley's algorithm for the above scenario. Assume that the clocks at the institutes are slave/clients.

2. Manipal Buddy is a banking and education application for the students and staff of MIT, Manipal. Mr Vinay, a sixth semester student wants to pay the end semester exams fees for a re-registered course. He simultaneously wishes to register for a course on NPTEL through the app. To register for exam he uses the mobile app whereas to register for NPTEL course he uses his laptop to log in. As he needs to finish both the registrations on the same day, he tries to do both the tasks simultaneously. Analyse and demonstrate using a program how Cristian's algorithm can be used in the above case to synchronize the clocks. Assume the relevant parameters.

3. Simulate a scenario in distributed systems to implement the Bully Algorithm for choosing a coordinator node amongst the participative nodes of the system after the collapse of the existing coordinator node in the system.

4. Simulate a scenario in distributed systems to implement the Ring Algorithm for choosing a coordinator node amongst the participative nodes of the system after the collapse of the existing coordinator node in the system.

References:

1. Mahajan, S. and Shah, S., 2015. *Distributed Computing*. 2nd ed. Oxford University Press, pp.141-209.
2. GeeksforGeeks. 2020. *Geeksforgeeks | A Computer Science Portal For Geeks*. [online] Available at: <https://www.geeksforgeeks.org/> [Accessed 3 April 2020].
3. American-time.com. 2020. *Food Manufacturing Clock Synchronization Case Study | American Time*. [online] Available at: <https://www.american-time.com/industries/manufacturing/case-studies/food-manufacturing> [Accessed 3 April 2020].
