Indian Institute of Information Technology Allahabad
Department of Information Technology

Distributed Systems

6^{th} Sem-2021

Project Report on
Distributed Chat System

Authors

- **Aadarsh Roshan Nandhakumar** (IIT2019161)
- **Ayush Baranwal** (IIT2019169)
- **Snehalreet Kaur** (IIT2019175)
- **Sahil Sharma** (IIT2019179)
- **Gaurav Nimrani** (IIT2019231)

I. Details of the implementation and Working

1) Basic Chat System

The chat system is a distributed python program based on MPI(message passing interface). It first elects a leader process using the Chang robertson algorithm. This leader process opens the text file and reads the list of processes participating in the chat. The list is then sent to the relevant members.

```
def leaderfunc():
    curline = 1
    data = linecache.getline('input.txt', curline)              #Leader first reads the file and retrieves the chatlist

    while(data):
        chatlist = data.split()
        if(len(chatlist) > size):                               #if chatlist is greater than world size, terminate
            print('World is smaller than total chat members!')
            break
        for i in range(len(chatlist)):                          #send chatlist and line to start from
                                                                #to all processes in chatlist
            if(not(int(chatlist[i]) == rank)):
                comm.send(data, dest=int(chatlist[i]))
                comm.send(curline+1, dest=int(chatlist[i]))
                time.sleep(1)

        if(str(rank) in chatlist):
            comm.isend(data, dest=int(chatlist[i]))             #If leader is a process in chatlist, send asynchronously
            comm.isend(curline+1, dest=int(chatlist[i]))
            managemessage()
```

Other processes receive the chat list and read the first line. If the first line is meant for the process, it starts sending messages, else it starts receiving messages.

```
    data = linecache.getline('input.txt', recline)
    recline = recline + 1

    if(int(data[0]) == rank):                                   #IF first line read is message to be sent by current
        print(data[0], 'sends', data[4:])
        for i in range(len(chatlist)):
            if(not(int(chatlist[i]) == rank)):
                addtosentcount(int(chatlist[i]))                #Add to receive count
        sendmessage(chatlist, recline)
    else:
        print(data)                                             #Receive data
        addtorecievecount(int(data[0]))                         #Add to receive count
        recvmessage(chatlist)
```

The sender process keeps sending messages till it gets to a message that is not its own. It then hands over control to the relevant process.

```
    if(not(int(data[0]) == rank)):                          # If data read does not belong to process
        comm.send(recline, dest=int(data[0]))               # Send line to start from to relevant process
        time.sleep(1)
        print('Handing chat to', data[0])
        recvmessage(chatlist)                               #start receiving messages
        break
    else:
        for i in range(len(chatlist)):                      #If data belong to process
            if(not(int(chatlist[i]) == rank)):
                print(rank, "sends", data[4:])                #Send to data
                comm.send(data, dest=int(chatlist[i]))
                time.sleep(1)
                addtosentcount(int(chatlist[i]))              #Add to sent count for process
    recline = recline + 1                                     #increment recline
```

The receiver process receives messages till its receives an integer, indicating that it needs to begin sending messages.

```
while(True):
    data = comm.recv(source=MPI.ANY_SOURCE)                 #Receive incoming data
    time.sleep(1)

    if(isinstance(data, int)):                             #If line number is received
        recline = int(data)
        sendmessage(chatlist, recline)                     #Start sending messages
        break
```

```
        else:
            addtorecievecount(int(data[0]))                 #Add received count
            print(data)                                     #Print received data
```

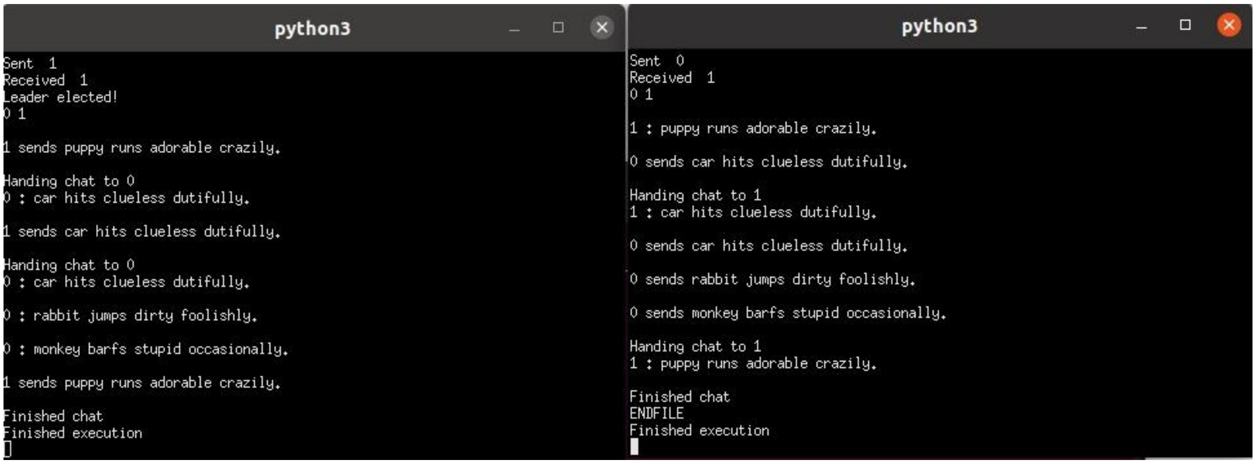
```
    if(isinstance(data, int)):                             #If line number is received
        recline = int(data)
        sendmessage(chatlist, recline)                     #Start sending messages
        break
```

When a process reads END in the file, it sends a message to the other processes in the chat, to end the chat. The other processes then initiate the chandy lamport snapshot algorithm.

```
    else:
        if(data[0] == 'E'):                                 #If END is received, finish chat and start receiving sna
            print('Finished chat')
            # break
```

When all processes receive ENDFILE from the leader, the processes terminate.

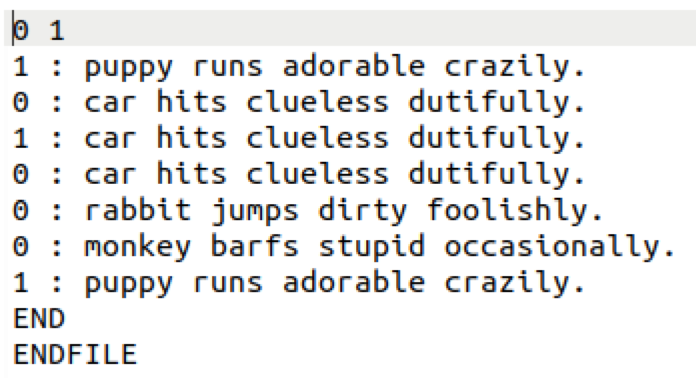
```
    print(reclist)
    if(reclist[0] == 'E'):                                  #If leader sends ENDFILE, finish execution
        print('Finished execution')
        break
```



```
Sent  1
Received  1
Leader elected!
0 1

1 sends puppy runs adorable crazily.

Handing chat to 0
0 : car hits clueless dutifully.

1 sends car hits clueless dutifully.

Handing chat to 0
0 : car hits clueless dutifully.

0 : rabbit jumps dirty foolishly.

0 : monkey barfs stupid occasionally.

1 sends puppy runs adorable crazily.

Finished chat
Finished execution
```

```
Sent  0
Received  1
0 1

1 : puppy runs adorable crazily.

0 sends car hits clueless dutifully.

Handing chat to 1
1 : car hits clueless dutifully.

0 sends car hits clueless dutifully.

0 sends rabbit jumps dirty foolishly.

0 sends monkey barfs stupid occasionally.

Handing chat to 1
1 : puppy runs adorable crazily.

Finished chat
ENDFILE
Finished execution
```

Output of two processes



```
0 1
1 : puppy runs adorable crazily.
0 : car hits clueless dutifully.
1 : car hits clueless dutifully.
0 : car hits clueless dutifully.
0 : rabbit jumps dirty foolishly.
0 : monkey barfs stupid occasionally.
1 : puppy runs adorable crazily.
END
ENDFILE
```

input.txt

Textfile generation :

A random number of processes to participate in the chat is chosen. Then a random sampling of processes is listed. A process is randomly chosen from the list and assigned a randomly generated sentence. The chatlist , and the respective chat is written to a text file.

2) Chandy Lamport Algorithm in chat system

The Chandy Lamport algorithm is used to record a consistent global snapshot for each of the process channels . This algorithm works using marker messages. Each process that wants to initiate a snapshot records its local state and sends a marker on each of its outgoing channels. All the other processes, upon receiving a marker, record their local state, the state of the channel from which the marker just came as empty, and send marker messages on all of their outgoing channels. If a process receives a marker after having recorded its local state, it records the state of the incoming channel from which the marker came as carrying all the messages received since it first recorded its local state.

Marker sending rule for each process:

1. Current process records its state.
2. For each outgoing channel where the marker has not been sent, send a marker across the channel before further message is sent.

Marker receiving rule for each process:

On receiving a marker along a channel C:

If current process has not recorded its state:

1. Record the state of C as the empty set
2. Execute the marker sending rule

Else:

1. Record the state of C as the set of messages received along C after the process state was recorded and before process received the marker along C.

Initiating a snapshot

- Process Pi initiates the snapshot
- Pi records its own state and prepares a special marker message.
- Send the marker message to all other processes.
- Start recording all incoming messages from channels Cij for j not equal to i.

Propagating a snapshot

- For all processes Pj consider a message on channel Ckj.
- If marker message is seen for the first time:
    - Pj Records own state and marks Ckj as empty
    - Send the marker message to all other processes

- Record all incoming messages from channels Clj for 1 not equal to j or k.
- Else add all messages from inbound channels.

Terminating a snapshot

- All processes have received a marker.
- All processes have received a marker on all the N-1 incoming channels.
- A central server can gather the partial state to build a global snapshot.

Implemented in code:

- Maps are used to store the count of process send and receive for each process which are initialized to 0.
- After each process sends a message addtosentcount function is called to increment the count in the map for the corresponding process and similarly addtorecievecount is called for each received message.

```
# functiin used to add messege count into process send count buffer


def addtosentcount(sendto):
    global messege_sent_count
    if not (sendto in messege_sent_count):
        messege_sent_count[sendto] = 1
    else:
        messege_sent_count[sendto] += 1

# functiin used to add messege count into process recieve count buffer


def addtorecievecount(recievedfrom):
    global messege_recieved_count
    if not (recievedfrom in messege_recieved_count):
        messege_recieved_count[recievedfrom] = 1
    else:
        messege_recieved_count[recievedfrom] += 1
```

- Ater a ENDILE is read from, the snapshot algorithm is initiated by the last process which sends the message by calling sendmarker function.
- The sendmarker function is called for initiating the algorithm or marker is received first time which stores the current snapshot of the process and sends the marker signal to another process.

```python
def sendmarker(chatlist):
    global marked, messege_sent_count_marker, messege_sent_count, messege_recieved_count, messege_recieved_count_marker
    # saving snapshot at current instant
    messege_sent_count_marker = messege_sent_count
    messege_recieved_count_marker = messege_recieved_count
    # sending marker to other processes
    for process in chatlist:
        if not (int(process) == rank):
            data = "||marker||"+" "+str(rank)
            comm.send(data, dest=int(process))
            time.sleep(1)
    marked = True
```

- And after the marker is received by any process it checks if it is received for first time. If Yes then call sendmarker function. Or Else store the messages count as messages are present in channel.

```python
# condition to recieve marker
elif data[0:10] == "||marker||":
    # recieved first time
    if marked == False:
        sendmarker(chatlist)
    # recievd other than first time so present in channel
    else:
        marker_from = int(data[11])
        messege_channel_marker[marker_from] = messege_recieved_count[marker_from] - \
            messege_recieved_count_marker[marker_from]
```

- After some time when the simulation of the algorithm is completed the initiator function calls the collectsnapshot function which stores its current snapshot and sends the collect signal to other processes.

```python
def collectsnapshot(chatlist):
    file = open("snapshot.txt", "w+")
    # sending signal to collect snapshot
    for process in chatlist:
        if not (int(process) == rank):
            data = "||collect||"+" "+str(rank)
            comm.send(data, dest=int(process))
            time.sleep(1)
    # storing the snapshot of initiator
    finalsnapshot = "Process " + str(rank) + " snapshot" + "\n"
    for count in messege_sent_count_marker.keys():
        finalsnapshot += "Sent " + \
            str(messege_sent_count_marker[count]) + \
            " messegaes to "+str(count)+"\n"

    for count in messege_recieved_count_marker.keys():
        finalsnapshot += "Recieved " + \
            str(messege_recieved_count_marker[count]
                )+" messeges from "+str(count)+"\n"

    for count in messege_channel_marker.keys():
        finalsnapshot += "Messeges in channel " + \
            str(rank)+" - "+str(count)+" : " + \
            str(messege_channel_marker[count])+"\n"
    finalsnapshot += "\n"
```

- Each process after receiving collect signal call sendsnapshot which stores its current snapshot in form of string and send it to initiator process which finally snapshot of all processes are stored in snapshot.txt

```python
def sendsnapshot(destination):
    # collecting snapshot of process in snapshot string
    snapshot = ""
    for count in messege_sent_count_marker.keys():
        snapshot += "Sent " + \
            str(messege_sent_count_marker[count]) + \
            " messegaes to "+str(count)+"\n"
    for count in messege_recieved_count_marker.keys():
        snapshot += "Recieved " + \
            str(messege_recieved_count_marker[count]
                )+" messeges from "+str(count)+"\n"
    for count in messege_channel_marker.keys():
        snapshot += "Messeges in channel " + \
            str(rank)+" - "+str(count)+" : " + \
            str(messege_channel_marker[count])+"\n"

    # sending snapshot to initiator process
    comm.send(snapshot, dest=destination)
    time.sleep(1)
```

```
Process 1 snapshot
Sent 4 messegaes to 0
Sent 0 messegaes to 1
Recieved 3 messeges from 0
Recieved 0 messeges from 1
Messeges in channel 1 - 0 : 0
Messeges in channel 1 - 1 : 0

Process 0 snapshot
Sent 0 messegaes to 0
Sent 3 messegaes to 1
Recieved 0 messeges from 0
Recieved 4 messeges from 1
Messeges in channel 0 - 0 : 0
Messeges in channel 0 - 1 : 0

```

3) Leader Election Algorithm

Leader Election Algorithm is mainly used to elect a leader/coordinator which will allot the tasks/messages to other processes . The leader cannot be manually selected as many processes may be leaving or joining the system . We have implemented the Chang - Robertson algorithm.

Chang Robertson algorithm:

It assumes that the processes are connected in a logical ring order.

Processes are able to skip faulty processes

1: Process P thinks that the coordinator has crashed,builds an election message containing its own id number.

2:Sends to the first live successor.

3:Each process adds its own number if it is greater than the number on the token and forwards it. It also colors itself red

```
color = 'red'   #initially process colour is red
tosend = rank+1 #process has to send to its immediate sucessive rank
if(tosend == size):       #if last process, connect to first to forma a circular ring.
    tosend = 0

if(rank == 0):
    token = 0
    comm.send(token, dest=tosend)              #send token with process number, 0 is the root process initialising
    print('Sent ', token)                      #leadership election
    time.sleep(1)
else:
    token = comm.recv(source=rank-1)           # receive token from preceeding process
    print('Received ', token)
    time.sleep(1)
    if(rank > token):                          #if current rank of process is greater than rank of token
        color = 'black'                        #process color's itself black
        token = rank                           #process assign itself as token
    comm.send(token, dest=tosend)              #send token to next process
    print('Sent ', token)
    time.sleep(1)
```

4: If the process is smaller than the number on the token, it just forwards it.

5:When the message returns to P, it sees its own process id in the list and knows that the circuit is completed and it is the leader.

6:Rest of the processes retrieve the leader process no. from the token when received for the second time

```
torecv = rank - 1
if(rank == 0):
    torecv = size - 1
token = comm.recv(source=torecv)               #receive token again
print('Received ', token)
time.sleep(1)
curleader = token
if(token == rank):                             #if token and rank are same, process is elected as leader
    print('Leader elected!')
    leader = True
comm.send(token, dest=tosend)                  #send token to next process
if(rank == 0):
    token = comm.recv(source=torecv)           #0 receives token and terminates
    time.sleep(1)
```

II.) Runtime Results , Experimental setup

No. of processors	Time required
2	22.08
3	26.03
4	28.07
5	38.06
16	200

CPU	Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz x 8
Memory	8 GB
OS	Ubuntu 20.04 LTS , 64 bit
MPI	mpiexec(OpenRTE) 4.0.3
Compiler	Python 3.8.10