**NETWORK LAB REPORT**

**NAME:** ANURAN CHAKRABORTY

**ROLL NO.:** 20

**CLASS:** BCSE-III

**SECTION:** A1

**ASSIGNMENT NUMBER:** 3

**PROBLEM STATEMENT:**

implement p-persistent CSMA with exponential backoff and additive backoff. Measure the performance parameters like throughput (i.e., average amount of data bits successfully transmitted per unit time) and forwarding delay (i.e., average end-to-- end delay, including the queuing delay and the transmission delay) experienced by the CSMA frames (IEEE 802.3). Plot the comparison graphs for throughput and forwarding delay by varying p. State your observations on the impact of different data rates for exponential/additive backoff along with p-persistent CSMA.

**DEADLINE:** 14TH MARCH, 2019

**SUBMITTED ON:** 14TH MARCH, 2019

**REPORT SUBMITTED ON:** 28TH MARCH, 2019

The report has two sections one for the CSMA protocol and the other for the CSMA/CD protocol.

**CSMA:**

**DESIGN**

The program implements the CSMA protocol. The program consists of 3 modules.

1. **sender.py**

This file contains the code to perform the work of the sender. Read from the input file, create the frame to be sent to the receiver and the send the frame to the channel process. This process also receives the acknowledgement sent by the receiver and accordingly resends the frame if ack is not received or is corrupt. The sender continuously senses the channel to check whether it is busy. If the channel is busy it waits else if it is idle it sends the frame with a probability p or it waits for a random period of time and then again checks.

1. **channel.py**

This is the channel process whose task are the following

1. Receive frame from sender
2. Send this frame to the receiver
3. A parallel thread sends a continuous signal to the sender stating whether the channel is free or not.
4. **rec.py**

This file contains the code to perform the work of the receiver. It receives the frame sent by the sender from the channel process.

1. **common.py**

This file contains some common function to be used by all processes.

All inter-process communication has been carried out by making use of the Python3 **socket**.

|  |
| --- |
|  |
| ***Fig. 1.*** *A brief outline of the program design of CSMA* |

**Fig. 1** gives a brief outline of the program.

Some important parameters for the design of the program are:

**Frame format:** The frame format used in the sender process is described as follows. The input data is split into frames of 4 bits each. This frame is then sent to the channel process.

**Assumption:** During the design one assumption that has been made is that the number of bits in the input file is a multiple of 4.

**Input format:** The input for the program is a text file consisting of a string of only 0s and 1s.

**Output format:** The program output simulates the CSMA protocol.

**IMPLEMENTATION**

The assignment has been implemented in Python3. The detailed description is given below.

***common.py:***

This module contains some commonly used function in the modules.

The port specifications are given below

portSenderReceive=11001

portSenderSend=11002

portReceiverReceive=11003

portReceiverSend=11004

frame\_size=4

*createSocket(port):*

This function creates a socket and binds it to a port

*# Function to create a socket and bind it to a port*

**def** createSocket(port):

s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind(('', port))

**return** s

*allowConn(port):*

Function to establish a connection with the port.

*# Function to receive a connection*

**def** allowConn(s):

s.listen(5)

c, addr=s.accept()

**return** c, addr

*createConn(port):*

Function to create a socket with a port and connect to it.

*# Function to create a socket and connect to it*

**def** createConn(port):

sock=socket.socket()

sock.connect(('',port))

**return** sock

*send\_frame(frame,c):*

Function to send a frame through a particular socket.

*# Function to send a frame*

**def** send\_frame(frame, c):

*# Send the frame to the other process*

c.send(frame.encode())

*prepare\_frame(frame,c):*

Function to prepare the frame for the sender given the frame number by converting it to binary and applying crc..

*# Function to prepare a frame*

**def** prepare\_frame(frame,sn):

frame=str(sn)+frame

*# CRC application*

crcframe=err.crc([frame], err.generator\_poly, frame\_size)

**return** crcframe[0]

*generateAck(rn):*

Function to generate acknowledgement for the receiver given the frame number by converting it to binary and applying crc.

*# Function to generate ack*

**def** generateAck(rn):

*# Generate crc appended code*

ack=bin(rn)[2:]

crcframe=err.crc([ack], err.generator\_poly, frame\_size)

**return** crcframe[0]

*readFile(filename, frame\_size):*

Function to read the input file and split into frames.

*# Function to read the file and split into frames*

**def** readfile(filename, frame\_size):

*# Open the file for reading*

f=open(filename,'r')

data=f.read()

*# Now split the data into frames*

list\_of\_frames=[data[i:i+frame\_size] **for** i **in** range(0, len(data), frame\_size)]

**return** list\_of\_frames

*ins\_error(frame, list\_of\_bits):*

Function to insert error at certain bit positions in the frame.

*# Function to introduce error*

**def** ins\_error(frame, list\_of\_bit):

new=list(frame)

*# Inserting error in the given bit position here*

**for** i **in** range(len(list\_of\_bit)):

**if**(new[list\_of\_bit[i]]=='0'):

new[list\_of\_bit[i]]='1'

**elif** (new[list\_of\_bit[i]]=='1'):

new[list\_of\_bit[i]]='0'

new=''.join(new)

**return** new

***sender.py:***

This is the code for the sender process.

**import** **common** **as** **co**

**import** **time**

**import** **socket**

**import** **random**

**import** **time**

**import** **threading**

isBusyChannel=0

sockSend=co.createConn(co.portSenderReceive) *# Socket to send data to channel*

time.sleep(1)

sockSignal=co.createConn(co.portSenderSignal) *# Socket to send data to channel*

**print**('Connected to channel')

probab=10

p=4

timeSlot=2

list\_of\_frames=co.readfile('input.txt', co.frame\_size)

list\_of\_frames.append('#')

sendThread=threading.Thread(target=send\_frame, args=(list\_of\_frames,)) *# create the sending thread*

senseThread=threading.Thread(target=sense\_medium) *# create the sending thread*

sendThread.start()

senseThread.start()

sendThread.join()

sendThread.join()

*send\_frame(list\_of\_frames):*

This function takes as its parameter a list of frames and sends all the frames to the channel. It checks whether the channel is busy by checking the **isBusyChannel** flag and accordingly follows the p-persistent strategy to send the frames.

*# Function to send frames*

**def** send\_frame(list\_of\_frames):

i=0

**while**(True):

*# Sense the channel and check if flag is 1 then dont send*

**if**(isBusyChannel==0): *# Channel is free*

*# Send the frame with a probability p*

pr=random.randint(0,probab)

**if**(pr<=p):

*# Send the frame*

**print**('Sending frame '+str(i))

co.send\_frame(list\_of\_frames[i], sockSend)

**if**(list\_of\_frames[i]!='#'):

i=i+1

time.sleep(3)

**else**:

**print**('Waiting '+str(timeSlot))

time.sleep(timeSlot)

**continue**

**else**: *# Channel is busy*

**print**('Channel busy')

time.sleep(2)

**continue**

*sense\_medium():*

This function continuously receives a signal from the channel process which is an indication of whether the channel is busy or not and accordingly sets the **isBusyChannel** flag. This function is run as a separate thread in the program.

*# Function to sense the medium*

**def** sense\_medium():

**global** isBusyChannel

**while**(True):

**if**(sockSignal.recv(1024).decode()=='1'):

*# Means channel is busy*

isBusyChannel=1

**else**:

*# Means channel is not busy*

isBusyChannel=0

***channel.py:***

This module implements the channel process. It first creates the appropriate sockets and then it goes into an infinite loop waiting for the sender to send. It receives the frame from the sender and keeps it in a buffer. The frames from the buffer is then sent to the receiver. The channel has actually 4 parallel threads running. One main thread continuously checks if any new sender has connected to the channel. Whenever a new sender connects to the channel it starts a new thread to service that sender. In a parallel thread the frames are sent to the receiver.

*# This is the channel process*

**import** **socket**

**import** **threading**

**import** **common** **as** **co**

**import** **time**

sockSenderRec=co.createSocket(co.portSenderReceive)

sockSenderSignal=co.createSocket(co.portSenderSignal)

sockReceiverSend=co.createSocket(co.portReceiverSend)

allow\_new\_conn()s

*allow\_new\_conn():*

This function checks if a new connection is established.

**def** allow\_new\_conn():

sockReceiverSend.listen(5)

receive, addrrec=sockReceiverSend.accept()

recThread=threading.Thread(target=send\_to\_receiver, args=[receive])

recThread.start()

**while**(True):

*# Wait for a connection*

sockSenderRec.listen(5)

c, addr=sockSenderRec.accept()

**print**('Connected to sender')

sockSenderSignal.listen(5)

signal,addrsignal=sockSenderSignal.accept()

*# Start a new thread for the sender*

sendThread=threading.Thread(target=receive\_from\_sender, args=[c,addr])

sendThread.start()

signalThread=threading.Thread(target=send\_signal, args=[signal, addrsignal])

signalThread.start()

*send\_to\_receiver(receive):*

This function sends the frames to the receiver.

*# Function to send to receiver*

**def** send\_to\_receiver(receive):

**while**(True):

*# If buffer not empty send the frame and clear buffer*

**if**(len(co.shared\_buffer)>0):

*# Send the frame*

**print**('Sending frame to receiver ')

time.sleep(4)

co.send\_frame(co.shared\_buffer[0], receive)

**del** co.shared\_buffer[0]

**else**:

**continue**

*receive\_from\_sender(c, addr):*

This function services (receives data from sender) each sender in separate threads.

*# Function to receive data from sender*

**def** receive\_from\_sender(c, addr):

*# Receive data from the sender and keep it in stored buffer*

**print**('Started new connection to '+str(addr))

**while**(True):

*# Receive data from sender*

frame=c.recv(1024).decode()

time.sleep(2)

co.shared\_buffer.append(frame)

**print**(co.shared\_buffer)

*send\_signal(s, saddr):*

This function continuously sends every sender the signal whether the channel is busy or not.

**def** send\_signal(s, saddr):

signal=0

**while**(True):

**if**(len(co.shared\_buffer)>=1): *# Channel is busy*

signal=1

**else**: *# Channel not busy*

signal=0

*# Send the signal via socket*

co.send\_frame(str(signal), s)

***rec.py***

This module is the receiver process.

*receive\_frame():*

This function receives frames from the channel process via a socket.

**import** **common** **as** **co**

**import** **socket**

sockReceive=co.createConn(co.portReceiverSend) *# Socket to send data to channel*

**print**('Connected to channel')

*# Function to receive a frame*

**def** receive\_frame():

**while**(True):

*# Receive the frame*

frame=sockReceive.recv(1024).decode()

**print**('Frame received '+frame)

receive\_frame()

**OUTPUTS**

|  |
| --- |
|  |

**RESULTS**

The throughput here was measured in terms of the attempts it took to send the entire data. With increase in p the number of collisions increased as more station tried to send data simultaneously. However it also sometimes decreased due to immediate sending of data.

**ANALYSIS**

Overall the implementation of the assignment is more or less correct. Some possible bugs can arise due to the assumption that the input size is a multiple of the frame size. However, this can easily be overcome by padding the last frame of the input data with 0s so that it is a multiple of the frame size. Currently the program works only for multiple sender and single receiver processes but the program may be modified to work with multiple sender and multiple receiver processes.

**COMMENTS**

Overall the lab assignment was a great learning experience as we got to implement the well-known CSMA protocol ourselves. The assignment can be rated as moderately difficult.

**CSMA/CD:**

**DESIGN**

The program implements the CSMA/CD protocol. The program consists of 3 modules.

1. **sender\_cd.py**

This file contains the code to perform the work of the sender. Read from the input file, create the frame to be sent to the receiver and the send the frame to the channel process. This process also receives the acknowledgement sent by the receiver and accordingly resends the frame if ack is not received or is corrupt. The sender continuously senses the channel to check whether it is busy. If the channel is busy it waits else if it is idle it sends the frame with a probability p or it waits for a random period of time and then again checks. Also, once the sender sends the frame it waits for a signal from the channel which states whether any collision was detected or not. If there is collision then the frame is resent.

1. **channel\_cd.py**

This is the channel process whose task are the following

1. Receive frame from sender
2. Send this frame to the receiver
3. Send jamming signal in case of collision
4. A parallel thread sends a continuous signal to the sender stating whether the channel is free or not.
5. **rec.py**

This file contains the code to perform the work of the receiver. It receives the frame sent by the sender from the channel process.

1. **common.py**

This file contains some common function to be used by all processes.

All inter-process communication has been carried out by making use of the Python3 **socket**.

|  |
| --- |
|  |
| ***Fig. 1.*** *A brief outline of the program design of stop and wait ARQ* |

**Fig. 1** gives a brief outline of the program.

Some important parameters for the design of the program are:

**Frame format:** The frame format used in the sender process is described as follows. The input data is split into frames of 4 bits each. This frame is then sent to the channel process.

**Assumption:** During the design one assumption that has been made is that the number of bits in the input file is a multiple of 4.

**Input format:** The input for the program is a text file consisting of a string of only 0s and 1s.

**Output format:** The program output simulates the CSMA/CD protocol.

**IMPLEMENTATION**

The assignment has been implemented in Python3. The detailed description is given below.

***common.py:***

This module contains some commonly used function in the modules.

The port specifications are given below

portSenderReceive=11001

portSenderSend=11002

portReceiverReceive=11003

portReceiverSend=11004

frame\_size=4

*createSocket(port):*

This function creates a socket and binds it to a port

*# Function to create a socket and bind it to a port*

**def** createSocket(port):

s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.bind(('', port))

**return** s

*allowConn(port):*

Function to establish a connection with the port.

*# Function to receive a connection*

**def** allowConn(s):

s.listen(5)

c, addr=s.accept()

**return** c, addr

*createConn(port):*

Function to create a socket with a port and connect to it.

*# Function to create a socket and connect to it*

**def** createConn(port):

sock=socket.socket()

sock.connect(('',port))

**return** sock

*send\_frame(frame,c):*

Function to send a frame through a particular socket.

*# Function to send a frame*

**def** send\_frame(frame, c):

*# Send the frame to the other process*

c.send(frame.encode())

*prepare\_frame(frame,c):*

Function to prepare the frame for the sender given the frame number by converting it to binary and applying crc..

*# Function to prepare a frame*

**def** prepare\_frame(frame,sn):

frame=str(sn)+frame

*# CRC application*

crcframe=err.crc([frame], err.generator\_poly, frame\_size)

**return** crcframe[0]

*generateAck(rn):*

Function to generate acknowledgement for the receiver given the frame number by converting it to binary and applying crc.

*# Function to generate ack*

**def** generateAck(rn):

*# Generate crc appended code*

ack=bin(rn)[2:]

crcframe=err.crc([ack], err.generator\_poly, frame\_size)

**return** crcframe[0]

*readFile(filename, frame\_size):*

Function to read the input file and split into frames.

*# Function to read the file and split into frames*

**def** readfile(filename, frame\_size):

*# Open the file for reading*

f=open(filename,'r')

data=f.read()

*# Now split the data into frames*

list\_of\_frames=[data[i:i+frame\_size] **for** i **in** range(0, len(data), frame\_size)]

**return** list\_of\_frames

*ins\_error(frame, list\_of\_bits):*

Function to insert error at certain bit positions in the frame.

*# Function to introduce error*

**def** ins\_error(frame, list\_of\_bit):

new=list(frame)

*# Inserting error in the given bit position here*

**for** i **in** range(len(list\_of\_bit)):

**if**(new[list\_of\_bit[i]]=='0'):

new[list\_of\_bit[i]]='1'

**elif** (new[list\_of\_bit[i]]=='1'):

new[list\_of\_bit[i]]='0'

new=''.join(new)

**return** new

***sender.py:***

This is the code for the sender process.

**import** **common** **as** **co**

**import** **time**

**import** **socket**

**import** **random**

**import** **time**

**import** **threading**

**if**(len(sys.argv)<2):

**print**('Error... usage python3 sender\_cd.py input\_file')

sys.exit();

filename=sys.argv[1]

isBusyChannel=0

sockSend=co.createConn(co.portSenderReceive) *# Socket to send data to channel*

time.sleep(1)

sockSignal=co.createConn(co.portSenderSignal) *# Socket to send data to channel*

**print**('Connected to channel')

probab=10

p=6

timeSlot=2

kmax=15

list\_of\_frames=co.readfile('input.txt', co.frame\_size)

list\_of\_frames.append('#')

sendThread=threading.Thread(target=send\_frame, args=(list\_of\_frames,)) *# create the sending thread*

senseThread=threading.Thread(target=sense\_medium) *# create the sending thread*

sendThread.start()

senseThread.start()

sendThread.join()

sendThread.join()

*send\_frame(list\_of\_frames):*

This function takes as its parameter a list of frames and sends all the frames to the channel. It checks whether the channel is busy by checking the **isBusyChannel** flag and accordingly follows the p-persistent strategy to send the frames. If collision is detected it resends the frame and increments max number of attempts. If the maximum number of attempts is reached it aborts the transmission of the frame.

*# Function to send frames*

**def** send\_frame(list\_of\_frames):

i=0

k=0

**while**(True):

*# Sense the channel and check if flag is 1 then dont send*

**if**(isBusyChannel==0): *# Channel is free*

*# Send the frame with a probability p*

pr=random.randint(0,probab)

**if**(pr<=p):

*# Send the frame*

**print**('Sending frame '+str(i))

co.send\_frame(list\_of\_frames[i], sockSend)

*# Wait for signal whether transmission successfull or collision*

isCollision=sockSend.recv(1024).decode()

**if**(isCollision=='0'): *# No collision successfull transmission*

**print**('Sent successfully')

**if**(list\_of\_frames[i]!='#'):

i=i+1

k=0

**else**:

*# If collision detected*

**print**('Frame collision detected..resending')

k=k+1

*# If max number of attempts exceeded*

**if**(k>kmax):

**print**('Max attempts for resending exceeded discarding frame')

i=i+1

k=0

**continue**

r=random.randint(0,2\*\*k-1)

waitTime=r\*timeSlot

time.sleep(waitTime)

time.sleep(3)

**else**:

**print**('Waiting '+str(timeSlot))

time.sleep(timeSlot)

**continue**

**else**: *# Channel is busy*

**print**('Channel busy')

time.sleep(2)

**continue**

*sense\_medium():*

This function continuously receives a signal from the channel process which is an indication of whether the channel is busy or not and accordingly sets the **isBusyChannel** flag. This function is run as a separate thread in the program.

*# Function to sense the medium*

**def** sense\_medium():

**global** isBusyChannel

**while**(True):

**if**(sockSignal.recv(1024).decode()=='1'):

*# Means channel is busy*

isBusyChannel=1

**else**:

*# Means channel is not busy*

isBusyChannel=0

***channel.py:***

This module implements the channel process. It first creates the appropriate sockets and then it goes into an infinite loop waiting for the sender to send. It receives the frame from the sender and keeps it in a buffer. The frames from the buffer is then sent to the receiver. The channel has actually 4 parallel threads running. One main thread continuously checks if any new sender has connected to the channel. Whenever a new sender connects to the channel it starts a new thread to service that sender. In a parallel thread the frames are sent to the receiver.

*# This is the channel process*

**import** **socket**

**import** **threading**

**import** **common** **as** **co**

**import** **time**

*# shared\_buffer=[]*

sockSenderRec=co.createSocket(co.portSenderReceive)

sockSenderSignal=co.createSocket(co.portSenderSignal)

sockReceiverSend=co.createSocket(co.portReceiverSend)

threadLock=threading.Lock()

*allow\_new\_conn():*

This function checks if a new connection is established.

**def** allow\_new\_conn():

sockReceiverSend.listen(5)

receive, addrrec=sockReceiverSend.accept()

recThread=threading.Thread(target=send\_to\_receiver, args=[receive])

recThread.start()

**while**(True):

*# Wait for a connection*

sockSenderRec.listen(5)

c, addr=sockSenderRec.accept()

**print**('Connected to sender')

sockSenderSignal.listen(5)

signal,addrsignal=sockSenderSignal.accept()

*# Start a new thread for the sender*

sendThread=threading.Thread(target=receive\_from\_sender, args=[c,addr])

sendThread.start()

signalThread=threading.Thread(target=send\_signal, args=[signal, addrsignal])

signalThread.start()

*send\_to\_receiver(receive):*

This function sends the frames to the receiver.

*# Function to send to receiver*

**def** send\_to\_receiver(receive):

**while**(True):

*# If buffer not empty send the frame and clear buffer*

**if**(len(co.shared\_buffer)>0):

time.sleep(10)

**if**(len(co.shared\_buffer)==1):

*# Send the frame*

**print**('Sending frame to receiver ')

co.send\_frame(co.shared\_buffer[0], receive)

**del** co.shared\_buffer[0]

**else**:

**continue**

*receive\_from\_sender(c, addr):*

This function services (receives data from sender) each sender in separate threads. It also checks if the buffer has more than one frame in it (meaning collision) and sends a signal accordingly.

*# Function to receive data from sender*

**def** receive\_from\_sender(c, addr):

*# Receive data from the sender and keep it in stored buffer*

**print**('Started new connection to '+str(addr))

**while**(True):

*# Receive data from sender*

frame=c.recv(1024).decode()

threadLock.acquire()

co.shared\_buffer.append(frame)

**print**(co.shared\_buffer)

threadLock.release()

time.sleep(10)

**if**(len(co.shared\_buffer)>1):

*# Channel is busy*

co.send\_frame('1',c)

*# Clear the list*

co.shared\_buffer.clear()

**else**:

*# Channel not busy*

co.send\_frame('0',c)

*send\_signal(s, saddr):*

This function continuously sends every sender the signal whether the channel is busy or not.

**def** send\_signal(s, saddr):

signal=0

**while**(True):

**if**(len(co.shared\_buffer)>=1): *# Channel is busy*

signal=1

**else**: *# Channel not busy*

signal=0

*# Send the signal via socket*

co.send\_frame(str(signal), s)

***rec.py***

This module is the receiver process.

*receive\_frame():*

This function receives frames from the channel process via a socket.

**import** **common** **as** **co**

**import** **socket**

sockReceive=co.createConn(co.portReceiverSend) *# Socket to send data to channel*

**print**('Connected to channel')

*# Function to receive a frame*

**def** receive\_frame():

**while**(True):

*# Receive the frame*

frame=sockReceive.recv(1024).decode()

**print**('Frame received '+frame)

receive\_frame()

**OUTPUTS**

|  |
| --- |
|  |
|  |

**RESULTS**

The throughput here was measured in terms of the attempts it took to send the entire data. With increase in p the number of collisions increased as more station tried to send data simultaneously. However, it also sometimes decreased due to immediate sending of data.

**ANALYSIS**

Overall the implementation of the assignment is more or less correct. Some possible bugs can arise due to the assumption that the input size is a multiple of the frame size. However, this can easily be overcome by padding the last frame of the input data with 0s so that it is a multiple of the frame size. Currently the program works only for multiple sender and single receiver processes but the program may be modified to work with multiple sender and multiple receiver processes.

**COMMENTS**

Overall the lab assignment was a great learning experience as we got to implement the well-known CSMA/CD protocol ourselves. The assignment can be rated as moderately difficult.