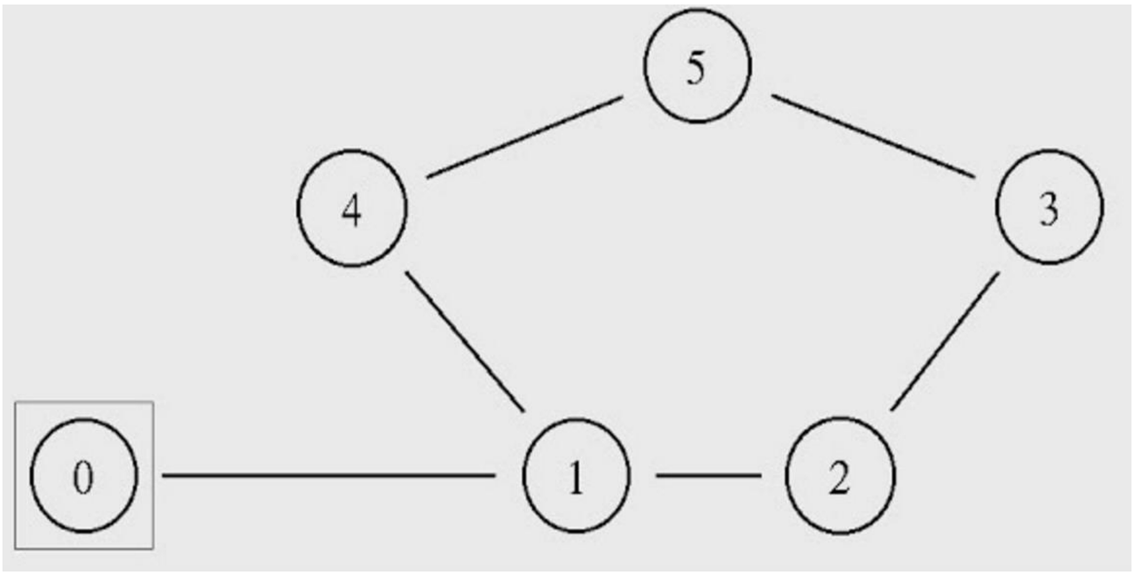
Exercise 1: Understanding the Impact of Network Dynamics on Routing

(include in your report)

In this exercise, we will observe how routing protocols react when network conditions change (e.g., a network link fails) using a ns-2 simulation.

The provided script, tp_routing.tcl takes no arguments and generates the network topology shown in the figure below.



You can run the simulation with the following command:

```
$ns tp_routing.tcl
```

Step 1: Run the script and observe the NAM window output.

Question 1. Which nodes communicate with which other nodes? Which route do the packets follow? Does it change over time?

Answer: Node 0 sends packets to Node 5, the transmitted packets follow the route 0-1-4-5, and the route does not change over time.

Node 2 sends packets to Node 5, the transmitted packets follow the route 2-3-5, and its route does not change either.

Note: You can also answer the above question by examining the simulation setting in the script file.

Step 2: Modify the script by uncommenting the following two lines (line No 84 and 85):

```
$ns rtmodel-at 1.0 down $n1 $n4

$ns rtmodel-at 1.2 up $n1 $n4
```

Step 3: Rerun the simulation and observe the NAM window output.

NOTE: Ignore the NAM syntax warnings on the terminal. These will not affect the simulation.

Question 2: What happens at time 1.0 and at time 1.2? Does the route between the communicating nodes change as a result of that?

Answer: At time 1.0, link 1-4 goes down, but the route between Node 1 and Node 5 does not change. And the Node 0 can't reach Node 5 at that time.

At time 1.2, link 1-4 goes up, and the packets can be forwarded again, and Node 0 can reach Node 5 again.

Step 4: The nodes in the simulation above use a static routing protocol (i.e., preferred routes do not change over time). We are going to change that, so that they use a Distance-Vector routing protocol. Modify the script and uncomment the following line (Line No 16) before the definition of the `finish` procedure.

```
$ns rtproto DV
```

Step 5: Rerun the simulation and observe the NAM window output.

Question 3: Did you observe any additional traffic as compared to Step 3 above? How does the network react to the changes that take place at time 1.0 and time 1.2 now?

Answer: Now, when link 1-4 goes down, the DV routing protocol learns the change of the network and changes the route to 0-1-2-3-5. However, when the link 1-4 goes up again, the routing protocol makes the route become 0-1-4-5 again. This is because the original route (0-1-4-5) has a lower cost (the number of hops to destination.)

Step 6: Comment the two lines (Lines 84 and 85) that you had added to the script in Step 2 and uncomment the following line (Line 87) instead:

```
$ns cost $n1 $n4 3
```

Step 7: Rerun the simulation and observe the NAM window output.

Question 4: How does this change affect the routing? Explain why.

Answer: This code changes the cost of link 1-4 to 3 (the initial cost of links is 1). And the route becomes 0-1-2-3-5, but not the 0-1-4-5. This is because the cost (4) of route 0-1-2-3-5 is lower than the cost (5) of route 0-1-4-5.

Step 8: Comment line 87 and Uncomment the following lines (Lines 89 and 90):

```
$ns cost $n1 $n4 2

$ns cost $n3 $n5 3
```

and uncomment the following (Line 29), which is located right after the finish procedure definition:

```
Node set multiPath_ 1
```

Step 9: Rerun the simulation and observe the NAM window output.

Question 5: Describe what happens and deduce the effect of the line you just uncommented.

Answer: The number of route from Node 2 to Node 5 is two, one is 2-3-5 and the other is 2-1-4-5. Since the cost of link 1-4 is changed to 2, and the cost of link 3-5 is changed to 3, the cost of these two routes has equal cost to destination. In addition, the "Node set multiPath_ 1" makes the network use multipath routing, so the data from node 2 will split traffic equally on both these two paths.

Exercise 3: Understanding IP Fragmentation

(Include in your report)

We will try to find out what happens when IP fragments a datagram by increasing the size of a datagram until fragmentation occurs. You are provided with a Wireshark trace file ip_frag that contains trace of sending pings with specific payloads to 8.8.8.8. We have used ping with option (– s option on Linux) to set the size of data to be carried in the ICMP echo request message. Note that the default packet size is 64 bytes in Linux (56 bytes data + 8 bytes ICMP header). Also note that Linux implementation for ping also uses 8 bytes of ICMP time stamp option leaving 48 bytes for the user data in the default mode. Once you have send a series of packets with the increasing data sizes, IP will start fragmenting packets that it cannot handle. We have used the following commands to generate this trace file.

Step 1: Ping with default packet size to the target destination as 8.8.8.8

```
ping -c 10 8.8.8.8
```

Step 2: Repeat by sending a set of ICMP requests with data of 2000.

```
ping -s 2000 -c 10 8.8.8.8
```

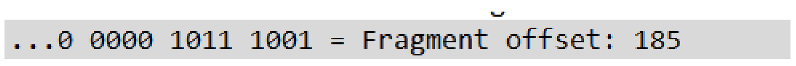
Step 3: Repeat again with data size set as 3500

```
ping -s 3500 -c 10 8.8.8.8
```

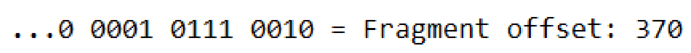
Load this trace file in Wireshark, filter on protocol field ICMP (you may need to clear the filter to see the fragments) and answer the following questions.

Question 1: Which data size has caused fragmentation and why? Which host/router has fragmented the original datagram? How many fragments have been created when data size is specified as 2000?

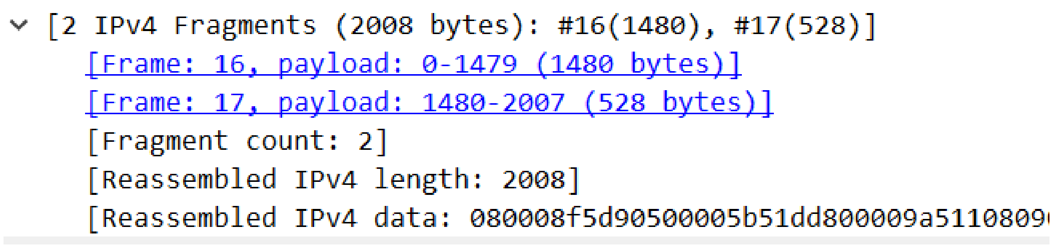
Answer: The data size 2000 and 3500 caused fragmentation since their 'offset' is set like below. That means the data is separated and 'offset' is a flag to recombine the data. The 192.168.1.103 fragmented the original datagram. In data size of 2000, there are two fragments created.



```
...0 0000 1011 1001 = Fragment offset: 185
```



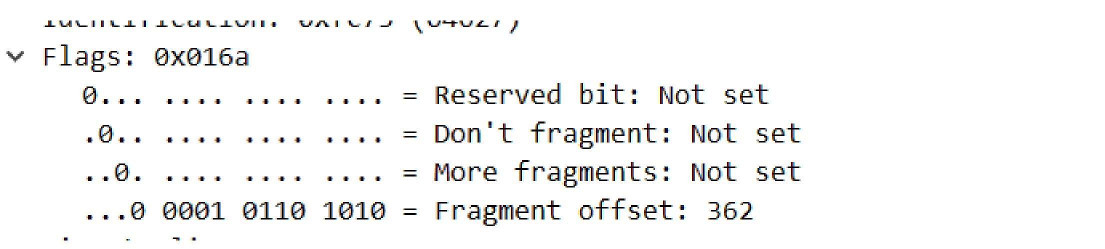
```
...0 0001 0111 0010 = Fragment offset: 370
```



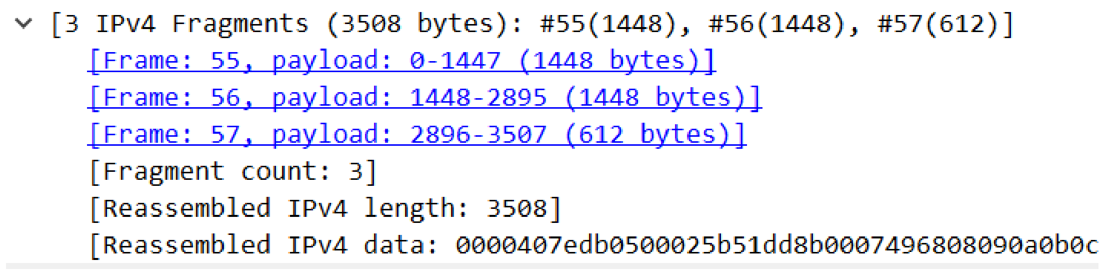
```
[2 IPv4 Fragments (2008 bytes): #16(1480), #17(528)]
    [Frame: 16, payload: 0-1479 (1480 bytes)]
    [Frame: 17, payload: 1480-2007 (528 bytes)]
    [Fragment count: 2]
    [Reassembled IPv4 length: 2008]
    [Reassembled IPv4 data: 080008f5d90500005b51dd800009a5110809
```

Question 2: Did the reply from the destination 8.8.8.8. for 3500-byte data size also get fragmented? Why and why not?

Answer: The reply from 8.8.8.8 for 3500-byte data size also gets fragmented. Like the below, we can find that the ICMP reply has the same data length with the ICMP request. Therefore, the data size is too large to transmit and it will be fragmented.



```
Flags: 0x016a
   0... .... .... .... = Reserved bit: Not set
   .0.. .... .... .... = Don't fragment: Not set
   ..0. .... .... .... = More fragments: Not set
   ...0 0001 0110 1010 = Fragment offset: 362
```



```
[3 IPv4 Fragments (3508 bytes): #55(1448), #56(1448), #57(612)]
   [Frame: 55, payload: 0-1447 (1448 bytes)]
   [Frame: 56, payload: 1448-2895 (1448 bytes)]
   [Frame: 57, payload: 2896-3507 (612 bytes)]
   [Fragment count: 3]
   [Reassembled IPv4 length: 3508]
   [Reassembled IPv4 data: 0000407edb0500025b51dd8b0007496808090a0b0c
```

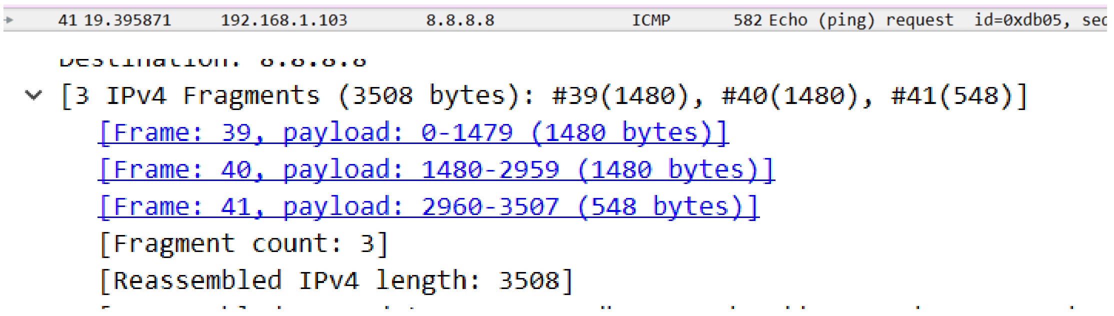
Question 3: Give the ID, length, flag and offset values for all the fragments of the first packet sent by 192.168.1.103 with data size of 3500 bytes?

Answer: The first packet sent by 192.168.1.103 with data size of 3500 bytes has this information respectively:

ID 0xdb05, length 1480, flag 1, offset 000

ID 0xdb05, length 1480, flag 1, offset 185

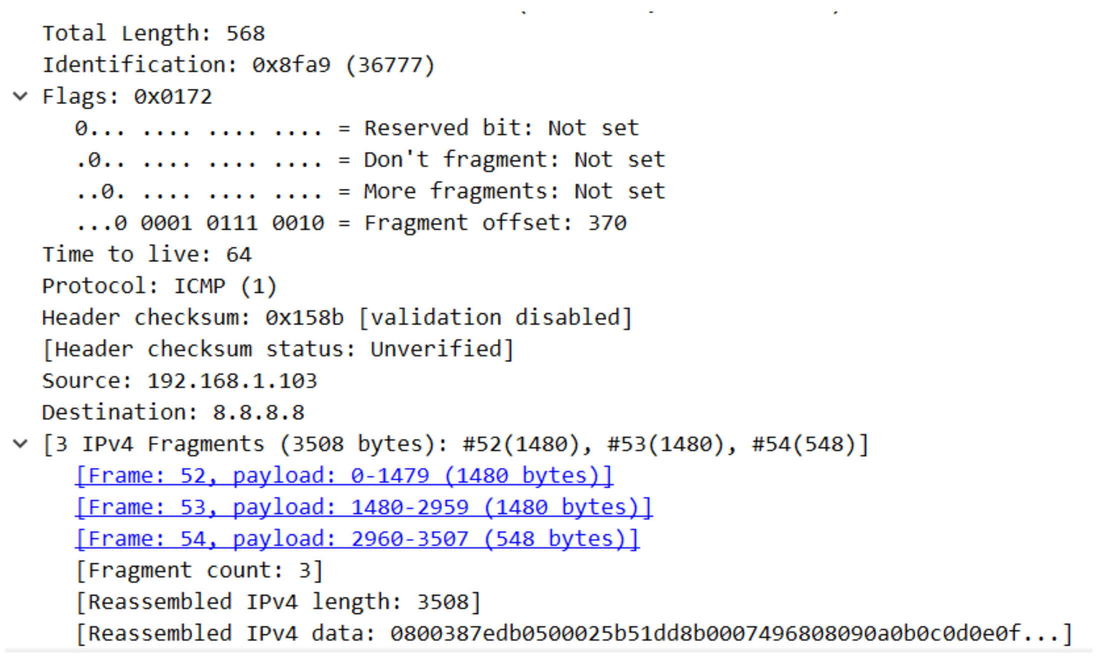
ID 0xdb05, length 548, flag 0, offset 370



```
41 19.395871    192.168.1.103    8.8.8.8    ICMP    582 Echo (ping) request  id=0xdb05, sec

[3 IPv4 Fragments (3508 bytes): #39(1480), #40(1480), #41(548)]
   [Frame: 39, payload: 0-1479 (1480 bytes)]
   [Frame: 40, payload: 1480-2959 (1480 bytes)]
   [Frame: 41, payload: 2960-3507 (548 bytes)]
   [Fragment count: 3]
   [Reassembled IPv4 length: 3508]
```

Question 4: Has fragmentation of fragments occurred when data of size 3500 bytes has been used? Why and why not?

Answer: We can't be sure. From data from Wireshark, we can only observe the situation when offset = 370 and length = 568 (548 + 20 IP header), which means that at least in the last packet of the fragment, no further fragmentation occurs. But we're not sure what happens when offset = 000 and offset = 185, so I don't think we can determine whether it is fragmented. If you want to know whether there is any fragmentation, there should have offset value which is 0<offset<185 or 185<offset<370.



```
    Total Length: 568
    Identification: 0x8fa9 (36777)
  ∨ Flags: 0x0172
      0... .... .... .... = Reserved bit: Not set
      .0.. .... .... .... = Don't fragment: Not set
      ..0. .... .... .... = More fragments: Not set
      ...0 0001 0111 0010 = Fragment offset: 370
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0x158b [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.1.103
    Destination: 8.8.8.8
  ∨ [3 IPv4 Fragments (3508 bytes): #52(1480), #53(1480), #54(548)]
      [Frame: 52, payload: 0-1479 (1480 bytes)]
      [Frame: 53, payload: 1480-2959 (1480 bytes)]
      [Frame: 54, payload: 2960-3507 (548 bytes)]
      [Fragment count: 3]
      [Reassembled IPv4 length: 3508]
      [Reassembled IPv4 data: 0800387edb0500025b51dd8b0007496808090a0b0c0d0e0f...]
```

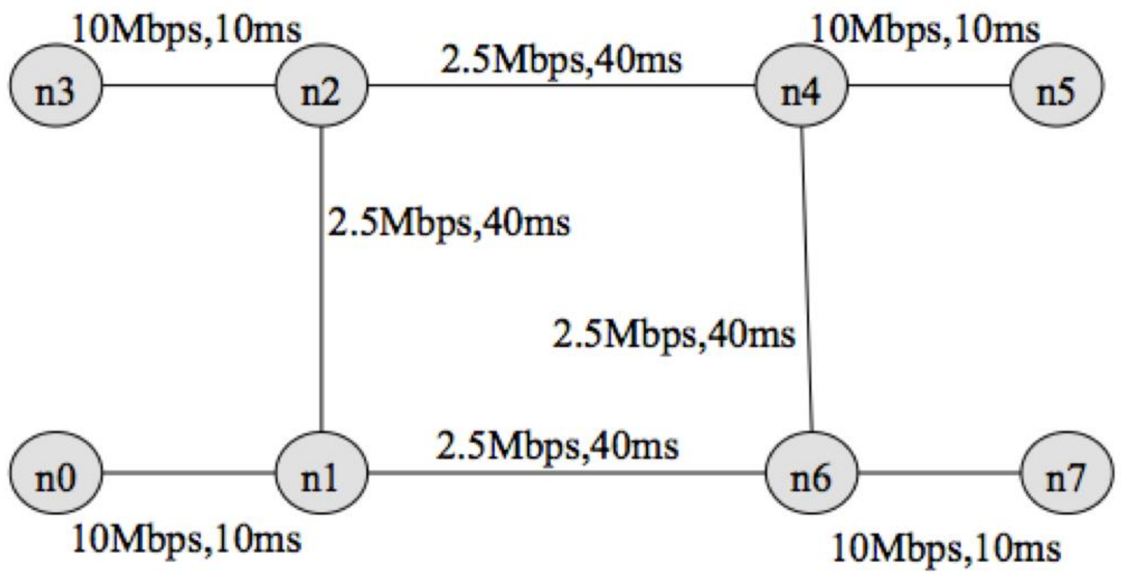
Question 5: What will happen if for our example one fragment of the original datagram from 192.168.1.103 is lost?

Answer: This causes the entire IP packet to be discarded, and if retransmission is required, it is implemented by the upper layer protocol of the network layer (IP). Therefore, we should try to avoid IP fragmentation so as to avoid unnecessary multiple transmission.

(*) Exercise 2: Setting up NS2 simulation for measuring TCP throughput

(Include in your report and demonstrate to your tutor)

Consider the topology shown in the following figure where bandwidth and delay for each link is shown.



You have been provided with a stub tcl file exercise2.tcl . Your task is to complete the stub file so that it runs with ns and produces two trace files tcp1.tr and tcp2.tr and nam.out. Check the animation for the simulation using nam.out file. Next write a script named "throughput.plot" (referenced from within exercise2.tcl in procedure finish( )) to plot the throughput received by host n5 for two flows terminating at n5. Uncomment the line (#exec gnuplot throughput.plot &) to execute gnuplot. You have been provided with the throughput plot TCPThroughput.png produced by gnuplot for comparing your final output.

">>" in the stub file indicates that one (or more) lines need to be added. Remove the ">>" and insert the required code.

Consider the following traffic pattern for your simulation.

FTP/TCP Source n0 -> TCP Sink n5 : start time: 0.5 sec End time: 8.5 sec

FTP/TCP Source n3 -> TCP Sink n5 : start time: 2.0 sec End time: 9.5 sec

FTP/TCP Source n7 -> TCP Sink n0 : start time: 3.0 sec End time: 9.5 sec

FTP/TCP Source n7 -> TCP Sink n3 : start time: 4.0 sec End time: 7.0 sec

You have to submit your completed tcl file (exercise2.tcl) and the script (throughput.plot) for producing the throughput plot. You should be prepared to comment on the throughput behaviour observed in the simulation.

exercise2.tcl

```
#Create a simulator object

set ns [new Simulator]
```

```
#Define different colors

$ns color 1 Blue

$ns color 2 Red

$ns color 3 Yellow




#Open the nam trace file

set namf [open out.nam w]

$ns namtrace-all $namf


#Open two trace files

set f1 [open tcp1.tr w]

set f2 [open tcp2.tr w]




#Define a 'finish' procedure

proc finish {} {

      global ns namf f1 f2

      $ns flush-trace

       #Close the trace amd nam files

      close $namf

       close $f1

       close $f2

      #Execute nam on the trace file

      exec nam out.nam &

      # Execute gnuplot to display the two trace files tcp1.tr and tcp2.tr

      exec gnuplot throughput.plot &

      exit 0

}
```

```
#Create eight nodes

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

set n3 [$ns node]

set n4 [$ns node]

set n5 [$ns node]

set n6 [$ns node]

set n7 [$ns node]



#Create links between the nodes

$ns duplex-link $n0 $n1 10Mb 10ms DropTail

$ns duplex-link $n3 $n2 10Mb 10ms DropTail

$ns duplex-link $n4 $n5 10Mb 10ms DropTail

$ns duplex-link $n6 $n7 10Mb 10ms DropTail

$ns duplex-link $n1 $n2 2.5Mb 40ms DropTail

$ns duplex-link $n1 $n6 2.5Mb 40ms DropTail

$ns duplex-link $n2 $n4 2.5Mb 40ms DropTail

$ns duplex-link $n6 $n4 2.5Mb 40ms DropTail


# set the correct orientation for all nodes

$ns duplex-link-op $n0 $n1 orient right

$ns duplex-link-op $n1 $n6 orient right

$ns duplex-link-op $n6 $n7 orient right

$ns duplex-link-op $n3 $n2 orient right

$ns duplex-link-op $n2 $n4 orient right

$ns duplex-link-op $n4 $n5 orient right

$ns duplex-link-op $n1 $n2 orient up

$ns duplex-link-op $n6 $n4 orient up
```

```
#Set Queue limit and Monitor the queue for the link between node 2 and node 4

$ns queue-limit $n2 $n4 10

$ns duplex-link-op $n2 $n4 queuePos 0.5


#Create a TCP agent and attach it to node n0

set tcp1 [new Agent/TCP]

$tcp1 set fid_ 1

$ns attach-agent $n0 $tcp1


#Sink for traffic at Node n5

set sink1 [new Agent/TCPSink]

$ns attach-agent $n5 $sink1


#Connect

$ns connect $tcp1 $sink1


#Setup FTP over TCP connection

set ftp1 [new Application/FTP]

$ftp1 attach-agent $tcp1


#Create a TCP agent and attach it to node n3

set tcp2 [new Agent/TCP]

$tcp2 set fid_ 2

$ns attach-agent $n3 $tcp2


#Sink for traffic at Node n5

set sink2 [new Agent/TCPSink]

$ns attach-agent $n5 $sink2
```

```
#Connect

$ns connect $tcp2 $sink2


#Setup FTP over TCP connection

set ftp2 [new Application/FTP]

$ftp2 attach-agent $tcp2


#Create a TCP agent and attach it to node n7

set tcp3 [new Agent/TCP]

$ns attach-agent $n7 $tcp3


#Sink for traffic at Node n0

set sink3 [new Agent/TCPSink]

$ns attach-agent $n0 $sink3


#Connect

$ns connect $tcp3 $sink3

#$tcp3 set fid_ 3


#Setup FTP over TCP connection

set ftp3 [new Application/FTP]

$ftp3 attach-agent $tcp3


#Create a TCP agent and attach it to node n7

set tcp4 [new Agent/TCP]

$ns attach-agent $n7 $tcp4


#Sink for traffic at Node n3

set sink4 [new Agent/TCPSink]

$ns attach-agent $n3 $sink4
```

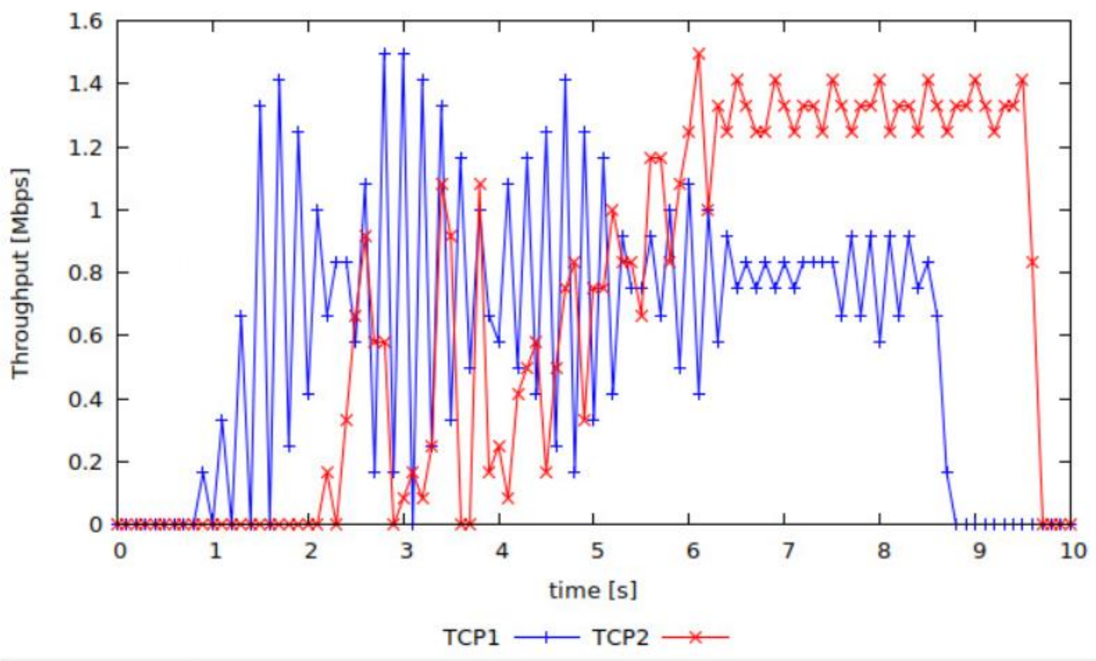
```
#Connect

$ns connect $tcp4 $sink4

$tcp4 set fid_ 3


#Setup FTP over TCP connection

set ftp4 [new Application/FTP]

$ftp4 attach-agent $tcp4


proc record {} {

    global ns f1 f2 sink1 sink2

        #Get an instance of the simulator

        set ns [Simulator instance]

        #Set the time after which the procedure should be called again

        set time 0.1

        #How many bytes have been received by the traffic sinks at n5?

        set bw1 [$sink1 set bytes_]

    set bw2 [$sink2 set bytes_]


    #Get the current time

        set now [$ns now]

        #Calculate the bandwidth (in MBit/s) and write it to the files

        puts $f1 "$now [expr $bw1/$time*8/1000000]"

        puts $f2 "$now [expr $bw2/$time*8/1000000]"

        #Reset the bytes_ values on the traffic sinks

    set bw1 [$sink1 set bytes_ 0]

    set bw2 [$sink2 set bytes_ 0]

        #Re-schedule the procedure

        $ns at [expr $now+$time] "record"

}
```

```
#Schedule events for all the agents

#start recording

$ns at 0.0 "record"


#start FTP sessions

$ns at 0.5 "$ftp1 start"

$ns at 2.0 "$ftp2 start"

$ns at 3.0 "$ftp3 start"

$ns at 4.0 "$ftp4 start"


#Stop FTP sessions

$ns at 8.5 "$ftp1 stop"

$ns at 9.5 "$ftp2 stop"

$ns at 9.5 "$ftp3 stop"

$ns at 7.0 "$ftp4 stop"


#Call the finish procedure after 10 seconds of simulation time

$ns at 10.0 "finish"


#Run the simulation

$ns run
```

throughput.plot

```
set xlabel "time [s]"
set ylabel "Throughput [Mbps]"
set key bel
plot    "tcp1.tr" u ($1):($2) t "TCP1" w lp lc 3, "tcp2.tr" u ($1):($2) t "TCP2" w lp lc 1
pause -1
```

We get the picture of TCP Throughput is below:


Throughput [Mbps]
time [s]
TCP1
TCP2