**COMPUTER NETWORKS LAB**

**KUMARAGURU COLLEGE OF TECHNOLOGY**

**Ex No: Study of NS2 and Congestion Control Algorithms in NS**

**Date :**

# Introduction

NS (Version 2) is an open source network simulation tool. It is an object oriented, discrete event driven simulator written in C++ and Otcl. The primary use of NS is in network researches to simulate various types of wired/wireless local and wide area networks; to implement network protocols such as TCP and UPD, traffic source behavior such as FTP, Telnet, Web, CBR and VBR, router queue management mechanism such as Drop Tail, RED and CBQ, routing algorithms such as Dijkstra, and many more.

Ns2 is written in C++ and Otcl to separate the control and data path implementations. The simulator supports a class hierarchy in C++ (the compiled hierarchy) and a corresponding hierarchy within the Otcl interpreter (interpreted hierarchy).

The reason why ns2 uses two languages is that different tasks have different requirements: For example simulation of protocols requires efficient manipulation of bytes and packet headers making the run-time speed very important. On the other hand, in network studies where the aim is to vary some parameters and to quickly examine a number of scenarios the time to change the model and run it again is more important.

In ns2, C++ is used for detailed protocol implementation and in general for such cases where every packet of a flow has to be processed. For instance, if you want to implement a new queuing discipline, then C++ is the language of choice. Otcl, on the other hand, is suitable for configuration and setup. Otcl runs quite slowly, but it can be changed very quickly making the construction of simulations easier. In ns2, the compiled C++ objects can be made available to the Otcl interpreter. In this way, the ready-made C++ objects can be controlled from the OTcl level.

## Otcl basics

This chapter introduces the syntax and some basic commands of the Otcl language that are used by ns2. It is important to understand how Otcl works before moving to the part that deals with the creation of the actual simulation scenario.

### Assigning values to variables

In tcl, values can be stored to variables and these values can be further used in commands:

**set a 5**

**set b [expr $a/5]**

In the first line, the variable a is assigned the value “5”. In the second line, the result of the command [expr $a/5], which equals 1, is then used as an argument to another command, which in turn assigns a value to the variable b. The “$” sign is used to obtain a value contained in a variable and square brackets are an indication of a command substitution.

### Procedures

One can define new procedures with the proc command. The first argument to proc is the name of the procedure and the second argument contains the list of the argument names to that procedure. For instance a procedure that calculates the sum of two numbers can be defined as follows:

**proc sum {a b} {**

**expr $a + $b**

**}**

The next procedure calculates the factorial of a number:

**proc factorial a {**

**if {$a <= 1} {**

**return 1**

**}**

**#here the procedure is called again**

**expr $x \* [factorial [expr $x-1]]**

**}**

It is also possible to give an empty string as an argument list. However, in this case the variables that are used by the procedure have to be defined as global. For instance:

**proc sum {} {**

**global a b**

**expr $a + $b**

**}**

### Files and lists

In tcl, a file can be opened for reading with the command:

**set testfile [open test.dat r]**

The first line of the file can be stored to a list with a command:

**gets $testfile list**

Now it is possible to obtain the elements of the list with commands (numbering of elements starts from 0) :

**set first [lindex $list 0]**

**set second [lindex $list 1]**

Similarly, a file can be written with a puts command:

**set testfile [open test.dat w]**

**puts $testfile “testi”**

### Calling subprocesses

The command exec creates a subprocess and waits for it to complete. The use of exec is similar to giving a command line to a shell program. For instance, to remove a file:

**exec rm $testfile**

The exec command is particularly useful when one wants to call a tcl-script from within another tclscript. For instance, in order to run the tcl-script example.tcl multiple times with the value of the parameter “test” ranging from 1 to 10, one can type the following lines to another tcl-script:

**for {set ind 1} {$ind <= 10} {incr ind} {**

**set test $ind**

**exec ns example.tcl test**

**}**

## Creating the topology

To be able to run a simulation scenario, a network topology must first be created. In ns2, the topology consists of a collection of nodes and links.

Before the topology can be set up, a new simulator object must be created at the beginning of the script with the command:

**set ns [new Simulator]**

The simulator object has member functions that enable creating the nodes and the links, connecting agents etc. All these basic functions can be found from the class Simulator. When using functions belonging to this class, the command begins with “$ns”, since ns was defined to be a handle to the Simulator object.

### Nodes

New node objects can be created with the command:

**set n0 [$ns node]**

**set n1 [$ns node]**

**set n2 [$ns node]**

**set n3 [$ns node]**

The member function of the Simulator class, called “node” creates four nodes and assigns them to the handles n0, n1, n2 and n3. These handles can later be used when referring to the nodes. If the node is not a router but an end system, traffic agents (TCP, UDP etc.) and traffic sources (FTP,CBR etc.) must be set up, i.e, sources need to be attached to the agents and the agents to the nodes, respectively.

### Agents, applications and traffic sources

The most common agents used in ns2 are UDP and TCP agents. In case of a TCP agent, several types are available. The most common agent types are:

* Agent/TCP – a Tahoe TCP sender
* Agent/TCP/Reno – a Reno TCP sender
* Agent/TCP/Sack1 – TCP with selective acknowledgement

The most common applications and traffic sources provided by ns2 are:

* Application/FTP – produces bulk data that TCP will send
* Application/Traffic/CBR – generates packets with a constant bit rate
* Application/Traffic/Exponential – during off-periods, no traffic is sent. During on-periods, packets are generated with a constant rate. The length of both on and off-periods is exponentially distributed.
* Application/Traffic/Trace – Traffic is generated from a trace file, where the sizes and interarrival times of the packets are defined.

In addition to these ready-made applications, it is possible to generate traffic by using the methods provided by the class Agent. For example, if one wants to send data over UDP, the method

**send(int nbytes)**

can be used at the tcl-level provided that the udp-agent is first configured and attached to some node.

Below is a complete example of how to create a CBR traffic source using UDP as transport protocol and attach it to node n0:

**set udp0 [new Agent/UDP]**

**$ns attach-agent $n0 $udp0**

**set cbr0 [new Application/Traffic/CBR]**

**$cbr0 attach-agent $udp0**

**$cbr0 set packet\_size\_ 1000**

**$udp0 set packet\_size\_ 1000**

**$cbr0 set rate\_ 1000000**

An FTP application using TCP as a transport protocol can be created and attached to node n1 in much the same way:

**set tcp1 [new Agent/TCP]**

**$ns attach-agent $n1 $tcp1**

**set ftp1 [new Application/FTP]**

**$ftp1 attach-agent $tcp1**

**$tcp1 set packet\_size\_ 1000**

The UDP and TCP classes are both child-classes of the class Agent. With the expressions [new Agent/TCP] and [new Agent/UDP] the properties of these classes can be combined to the new objects udp0 and tcp1. These objects are then attached to nodes n0 and n1. Next, the application is defined and attached to the transport protocol. Finally, the configuration parameters of the traffic source are set. In case of CBR, the traffic can be defined by parameters rate\_ (or equivalently interval\_, determining the interarrival time of the packets), packetSize\_ and random\_ . With the random\_ parameter it is possible to add some randomness in the interarrival times of the packets. The default value is 0, meaning that no randomness is added.

### Traffic Sinks

If the information flows are to be terminated without processing, the udp and tcp sources have to be connected with traffic sinks. A TCP sink is defined in the class Agent/TCPSink and an UDP sink is defined in the class Agent/Null.

A UDP sink can be attached to n2 and connected with udp0 in the following way:

**set null [new Agent/Null]**

**$ns attach-agent $n2 $null**

**$ns connect $udp0 $null**

A standard TCP sink that creates one acknowledgement per a received packet can be attached to n3 and connected with tcp1 with the commands:

**set sink [new Agent/Sink]**

**$ns attach-agent $n3 $sink**

**$ns connect $tcp1 $sink**

There is also a shorter way to define connections between a source and the destination with the command:

**$ns create-connection <srctype> <src> <dsttype> <dst> <pktclass>**

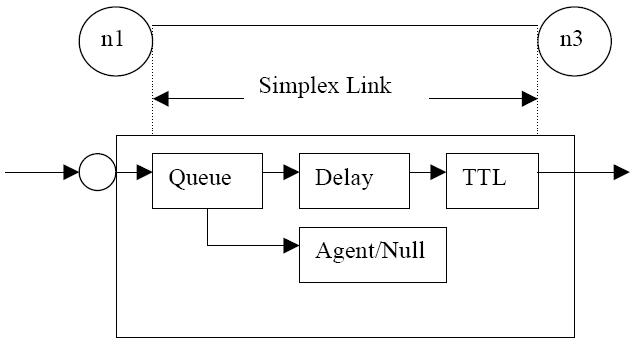
For example, to create a standard TCP connection between n1 and n3 with a class ID of 1:

**$ns create-connection TCP $n1 TCPSink $n3 1**

One can very easily create several tcp-connections by using this command inside a for-loop.

### Links

Links are required to complete the topology. In ns2, the output queue of a node is implemented as part of the link, so when creating links the user also has to define the queue-type.



**Figure 2 Link in ns2**

Figure 2 shows the construction of a simplex link in ns2. If a duplex-link is created, two simplex links will be created, one for each direction. In the link, packet is first enqueued at the queue. After this, it is either dropped, passed to the Null Agent and freed there, or dequeued and passed to the Delay object which simulates the link delay. Finally, the TTL (time to live) value is calculated and updated.

Links can be created with the following command:

**$ns duplex/simplex-link endpoint1 endpoint2 bandwidth delay queue-type**

For example, to create a duplex-link with DropTail queue management between n0 and n2:

**$ns duplex-link $n0 $n2 15Mb 10ms DropT ail**

Creating a simplex-link with RED queue management between n1 and n3:

**$ns simplex-link $n1 $n3 10Mb 5ms RED**

The values for bandwidth can be given as a pure number or by using qualifiers k (kilo), M (mega), b (bit) and B (byte). The delay can also be expressed in the same manner, by using m (milli) and u (mikro) as qualifiers. There are several queue management algorithms implemented in ns2, but in this exercise only DropTail and RED will be needed.

## Tracing and monitoring

In order to be able to calculate the results from the simulations, the data has to be collected somehow. NS2 supports two primary monitoring capabilities: traces and monitors. The traces enable recording of packets whenever an event such as packet drop or arrival occurs in a queue or a link. The monitors provide a means for collecting quantities, such as number of packet drops or number of arrived packets in the queue. The monitor can be used to collect these quantities for all packets or just for a specified flow (a flow monitor).

### Traces

All events from the simulation can be recorded to a file with the following commands:

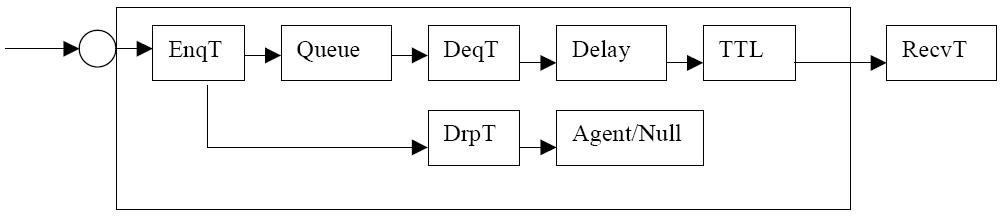
**set trace\_all [open all.dat w]**

**$ns trace-all $trace\_all**

**$ns flush-trace**

**close $trace\_all**

First, the output file is opened and a handle is attached to it. Then the events are recorded to the file specified by the handle. Finally, at the end of the simulation the trace buffer has to be flushed and the file has to be closed. This is usually done with a separate finish procedure. If links are created after these commands, additional objects for tracing (EnqT, DeqT, DrpT and RecvT) will be inserted into them.



**Figure 3 Link in ns2 when tracing is enabled**

These new objects will then write to a trace file whenever they receive a packet. The format of the trace file is following:

+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610

- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610

r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600

r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602

+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611

- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611

+ : enqueue

- : dequeue

d : drop

r : receive

The fields in the trace file are: type of the event, simulation time when the event occurred, source and destination nodes, packet type (protocol, action or traffic source), packet size, flags, flow id, source and destination addresses, sequence number and packet id. In addition to tracing all events of the simulation, it is also possible to create a trace object between a particular source and a destination with the command:

**$ns create-trace <type> <file> <src> <dest>**

where the type can be, for instance,

* Enque – a packet arrival (for instance at a queue)
* Deque – a packet departure (for instance at a queue)
* Drop – packet drop
* Recv – packet receive at the destination

Tracing all events from a simulation to a specific file and then calculating the desired quantities from this file for instance by using perl or awk and Matlab is an easy way and suitable when the topology is relatively simple and the number of sources is limited. However, with complex topologies and many sources this way of collecting data can become too slow. The trace files will also consume a significant amount of disk space.

### Monitors

With a queue monitor it is possible to track the statistics of arrivals, departures and drops in either bytes or packets. Optionally the queue monitor can also keep an integral of the queue size over time.

For instance, if there is a link between nodes n0 and n1, the queue monitor can be set up as follows:

**set qmon0 [$ns monitor-queue $n0 $n1]**

The packet arrivals and byte drops can be tracked with the commands:

**set parr [$qmon0 set parrivals\_]**

**set bdrop [$qmon0 set bdrops\_]**

Besides assigning a value to a variable the set command can also be used to get the value of a variable. For example here the set command is used to get the value of the variable “parrivals” defined in the queue monitor class.

A flow monitor is similar to the queue monitor but it keeps track of the statistics for a flow rather than for aggregated traffic. A classifier first determines which flow the packet belongs to and then passes the packet to the flow monitor.

The flowmonitor can be created and attached to a particular link with the commands:

**set fmon [$ns makeflowmon Fid]**

**$ns attach-fmon [$ns link $n1 $n3] $fmon**

Notice that since these commands are related to the creation of the flow-monitor, the commands are defined in the Simulator class, not in the Flowmonitor class. The variables and commands in the Flowmonitor class can be used after the monitor is created and attached to a link. For instance, to dump the contents of the flowmonitor (all flows):

**$fmon dump**

If you want to track the statistics for a particular flow, a classifier must be defined so that it selects the flow based on its flow id, which could be for instance 1:

**set fclassifier [$fmon classifier]**

**set flow [$fclassifier lookup auto 0 0 1]**

## Controlling the simulation

After the simulation topology is created, agents are configured etc., the start and stop of the simulation and other events have to be scheduled. The simulation can be started and stopped with the commands

**$ns at $simtime “finish”**

**$ns run**

The first command schedules the procedure finish at the end of the simulation, and the second command actually starts the simulation. The finish procedure has to be defined to flush the trace buffer, close the trace files and terminate the program with the exit routine. It can optionally start NAM (a graphical network animator), post process information and plot this information.

The finish procedure has to contain at least the following elements:

**proc finish {} {**

**global ns trace\_all**

**$ns flush-trace**

**close $trace\_all**

**exit 0**

**}**

Other events, such as the starting or stopping times of the clients can be scheduled in the following way:

**$ns at 0.0 “cbr0 start”**

**$ns at 50.0 “ftp1start”**

**$ns at $simtime “cbr0 stop”**

**$ns at $simtime “ftp1 stop”**

If you have defined your own procedures, you can also schedule the procedure to start for example every 5 seconds in the following way:

**proc example {} {**

**global ns**

**set interval 5**

**….**

**…**

**$ns at [expr $now + $interval] “example”**

**}**

**1.5 Congestion Control Mechanisms**

The essential strategy of TCP is to send packets into the network without a reservation and then to react to observable events that occur. TCP assumes only FIFO queuing in the network‘s routers, but also works with fair queuing.

* + 1. **Additive Increase/Multiplicative Decrease**

TCP maintains a new state variable for each connection, called Congestion Window, which is used by the source to limit how much data it is allowed to have in transit at a given time. The congestion window is congestion control‘s counterpart to flow control‘s advertised window. TCP is modified such that the maximum number of bytes of unacknowledged data allowed is now the minimum of the congestion window and the advertised window.

MaxWindow = MIN (CongestionWindow, AdvertisedWindow)

EffectiveWindow = MaxWindow − (LastByteSent − LastByteAcked).

That is, MaxWindow replaces AdvertisedWindow in the calculation of EffectiveWindow.

Thus, a TCP source is allowed to send no faster than the slowest component—the network or the destination host can accommodate. The problem, of course, is how TCP comes to learn an appropriate value for CongestionWindow. Unlike the AdvertisedWindow, sent by receiving side of the connection, there is no one to send a suitable CongestionWindow to the sending side of TCP. TCP does not wait for an entire window‘s worth of ACKs to add one packet‘s worth to the congestion window, but instead increments CongestionWindow by a little for each ACK that arrives. Specifically, the congestion window is incremented as follows each time an ACK arrives:

Increment = MSS × (MSS/CongestionWindow)

CongestionWindow + = Increment

That is, rather than incrementing CongestionWindow by an entire MSS bytes each RTT, we increment it by a fraction of MSS every time an ACK is received. The important concept to understand about AIMD is that the source is willing to reduce its congestion window at a much faster rate than it is willing to increase its congestion window.

**1.5.2** **Fast Retransmit and Fast Recovery**

The mechanisms described so far were part of the original proposal to add congestion control to TCP. It was soon discovered, however, that the coarse-grained implementation of TCP timeouts led to long periods of time during which the connection went dead while waiting for a timer to expire. Because of this, a new mechanism called fast re- transmit was added to TCP. Fast retransmit is a heuristic that sometimes triggers the retransmission of a dropped packet sooner than the regular timeout mechanism.

**1.6. Variants of TCP**

**1.6.1 TCP TAHOE**

TCP is based on a principle of conservation of packets‘, i.e. if the connection is running at the available bandwidth capacity then a packet is not injected into the network unless a packet

is taken out as well. TCP implements this principle by using the acknowledgements to clock outgoing packets because an acknowledgement means that a packet was taken off the wire by the receiver. It also maintains a congestion window CWD to reflect the network capacity. Tahoe suggests that whenever a TCP connection starts or re-starts after a packet loss it should go through a procedure called ‗slow-start‘. The reason for this procedure is that an initial burst

might overwhelm the network and the connection might never get started. The congestion window size is multiplicatively increased that is it becomes double for each transmission until it encounters congestion. Slow start suggests that the sender set the congestion window to 1 and then for each

ACK received it increase the CWD by 1. So in the first round trip time (RTT) we send 1 packet, in the second we send 2 and in the third we send 4. Thus we increase exponentially until we lose a packet which is a sign of congestion. When we encounter congestion we decreases our sending rate and we reduce congestion window to one. And start over again. The important thing is that Tahoe detects packet losses by timeouts. Sender is notified that congestion has occurred based on the packet loss.

**1.6.2 TCP RENO**

This RENO retains the basic principle of Tahoe, such as slow starts and the coarse grain retransmit timer. However it adds some intelligence over it so that lost packets are detected earlier and the pipeline is not emptied every time a packet is lost. Reno requires that we receive immediate acknowledgement whenever a segment is received. The logic behind this is that whenever we receive a duplicate acknowledgment, then his duplicate acknowledgment could have been received if the next segment in sequence expected, has been delayed in the network and the segments reached there out of order or else that the packet is lost. If we receive a number of duplicate acknowledgements then that means that sufficient time have passed and even if the segment had taken a longer path, it should have gotten to the receiver by now. There is a very high probability that it was lost.

So Reno suggests an algorithm called ‘Fast Re-Transmit’. Whenever we receive 3 duplicate ACK‘s we take it as a sign that the segment was lost, so we re-transmit the segment without waiting for timeout. Thus we manage to re-transmit the segment with the pipe almost full. Another modification that RENO makes is in that after a packet loss, it does not reduce the congestion window to 1. Since this empties the pipe. It enters into an algorithm which we call ‘Fast-Re-Transmit’.

**1.6.3 NEW-RENO**

New RENO is a slight modification over TCP-RENO. It is able to detect multiple packet losses and thus is much more efficient that RENO in the event of multiple packet losses. Like

RENO, New-RENO also enters into fast-retransmit when it receives multiple duplicate packets, however it differs from RENO in that it doesn‘t exit fast-recovery until all the data which was out standing at the time it entered fast recovery is acknowledged. The fast-recovery phase proceeds as in Reno, however when a fresh ACK is received then there are two cases:

* If it ACK‘s all the segments which were outstanding when we entered fast recovery then it exits fast recovery and sets CWD to threshold value and continues congestion avoidance like Tahoe.
* If the ACK is a partial ACK then it deduces that the next segment in line was lost and it retransmits that segment and sets the number of duplicate ACKS received to zero. It exits Fast recovery when all the data in the window is acknowledged.

**1.6.4 TCP SACK**

TCP with Selective Acknowledgments‘ is an extension of TCP RENO and it works around the problems face by TCP RENO and TCP New-RENO, namely detection of multiple lost packets, and re-transmission of more than one lost packet per RTT. SACK retains the slow-start and fast retransmits parts of RENO. It also has the coarse grained timeout of Tahoe to fall back on, in case a packet loss is not detected by the modified algorithm. SACK TCP requires that segments not be acknowledged cumulatively but should be acknowledged selectively. If there are no such segments outstanding then it sends a new packet**.** Thus more than one lost segment can be sent in one RTT.

**1.6.5 TCP FACK**

FACK or Forward Acknowledgement is a special algorithm that works on top of the SACK options, and is geared at congestion controlling. FACK algorithm uses information provided by SACK to add more precise control to the injection of data into the network during recovery – this is achieved by explicitly measuring the total number of bytes of data outstanding in the network. FACK decouples congestion control from data recovery thereby attaining more precise control over the data flow in the network. The main idea of FACK algorithm is to consider the most forward selective acknowledgement sequence number as a sign that all the previous un-(selectively)- acknowledged segments were lost. This observation allows improving recovery of losses significantly.

**1.6.6 TCP VEGAS**

VEGAS is a TCP implementation which is a modification of RENO. It builds on the fact that proactive measure to encounter congestion is much more efficient than reactive ones. It tried to get around the problem of coarse grain timeouts by suggesting an algorithm which checks for timeouts at a very efficient schedule. Also it overcomes the problem of requiring enough duplicate acknowledgements to detect a packet loss, and it also suggests a modified slow start

algorithm which prevents it from congesting the network. The three major changes induced by Vegas are:

**New Re-Transmission Mechanism:** Vegas extend on the re-transmission mechanism of RENO. It keeps track of when each segment was sent and it also calculates an estimate of the RTT by keeping track of how long it takes for the acknowledgment to get back.

**Congestion avoidance:** TCP Vegas is different from all the other implementation in its behavior during congestion avoidance. It does not use the loss of segment to signal that there is congestion. It determines congestion by a decrease in sending rate as compared to the expected rate, as result of large queues building up in the routers.

**Modified Slow-start:** TCP Vegas differs from the other algorithms during its slow-start phase. The reason for this modification is that when a connection first starts it has no idea of the available bandwidth and it is possible that during exponential increase it over shoots the bandwidth by a big amount and thus induces congestion. To this end Vegas increases exponentially only every other RTT, between that it calculates the actual sending through put to the expected and when the difference goes above a certain threshold it exits slow start and enters the congestion avoidance phase.

**RESULT:**

Thus, Congestion control algorithms were studied using NS2.

**Ex No: 4a USER DATAGRAM PROTOCOL USING NS-2**

**Date :**

**AIM:**

To implement User Datagram Protocol (UDP) using NS-2

**ALGORITHM:**

Step 1: Start network simulator OTCL editor.

Step 2: Create new simulator using set ns [new Simulator] syntax Step 3: Create procedure to trace all path

proc finish {} {

global ns nf tf

$ns flush-trace close $nf

close $tf

exec nam udp.nam & exit 0 }

Step 4: Connect with TCP and SINK command.

$ns connect $tcp $sink

Step 5: Run and Execute the program.

$ns run

**PROGRAM:**

set ns [new Simulator] set nf [open udp.nam w]

$ns namtrace-all $nf set tf [open out.tr w]

$ns trace-all $tf proc finish {} {

global ns nf tf

$ns flush-trace close $nf close $tf

exec nam udp.nam & exit 0

}

set n0 [$ns node] set n1 [$ns node] set n2 [$ns node] set n3 [$ns node] set n4 [$ns node] set n5 [$ns node]

$ns duplex-link $n0 $n4 1Mb 50ms DropTail

$ns duplex-link $n1 $n4 1Mb 50ms DropTail

$ns duplex-link $n2 $n5 0.1Mb 1ms DropTail

$ns duplex-link $n3 $n5 1Mb 1ms DropTail

$ns duplex-link $n4 $n5 1Mb 50ms DropTail

$ns duplex-link-op $n2 $n5 queuePos 1 set tcp [new Agent/UDP]

$ns attach-agent $n0 $tcp set sink [new Agent/Null]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

set ftp [new Application/Traffic/CBR]

$ftp attach-agent $tcp

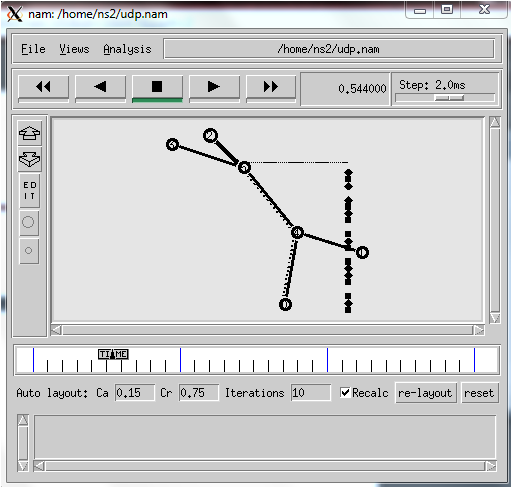
$ns at 0.0 "$ftp start"

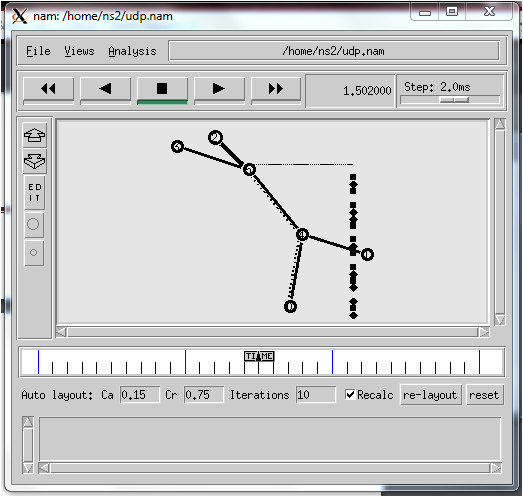
$ns at 2.5 "$ftp stop"

$ns at 3 "finish"

$ns run

**OUTPUT:**





**RESULT:**

Thus the program for implementing UDP was executed using NS-2 and output verified using Network Animator.

**Ex No: 4b TRANSMISSION CONTROL PROTOCOL USING NS-2**

**Date :**

**AIM:**

To implement Transmission Control Protocol (TCP) using NS-2.

**ALGORITHM:**

Step 1: Start network simulator OTCL editor.

Step 2: Create new simulator using set ns [new Simulator] syntax

Step 3: Create procedure to trace all path

proc finish {} {

global ns nf tf

$ns flush-trace close $nf

close $tf

exec nam tcp.nam & exit 0}

Step 4: Connect with TCP and SINK command.

$ns connect $tcp $sink

Step 5: Run and Execute the program.

$ns run

**PROGRAM:**

set ns [new Simulator] set nf [open tcp.nam w]

$ns namtrace-all $nf set tf [open out.tr w]

$ns trace-all $tf proc finish {} {

global ns nf tf

$ns flush-trace close $nf

close $tf

exec nam tcp.nam & exit 0

}

set n0 [$ns node] set n1 [$ns node] set n2 [$ns node] set n3 [$ns node] set n4 [$ns node] set n5 [$ns node]

$ns duplex-link $n0 $n4 1Mb 50ms DropTail

$ns duplex-link $n1 $n4 1Mb 50ms DropTail

$ns duplex-link $n2 $n5 1Mb 1ms DropTail

$ns duplex-link $n3 $n5 1Mb 1ms DropTail

$ns duplex-link $n4 $n5 1Mb 50ms DropTail

$ns duplex-link-op $n4 $n5 queuePos 0.5 set tcp [new Agent/TCP]

$ns attach-agent $n0 $tcp

set sink [new Agent/TCPSink]

$ns attach-agent $n2 $sink

$ns connect $tcp $sink

set ftp [new Application/FTP]

$ftp attach-agent $tcp

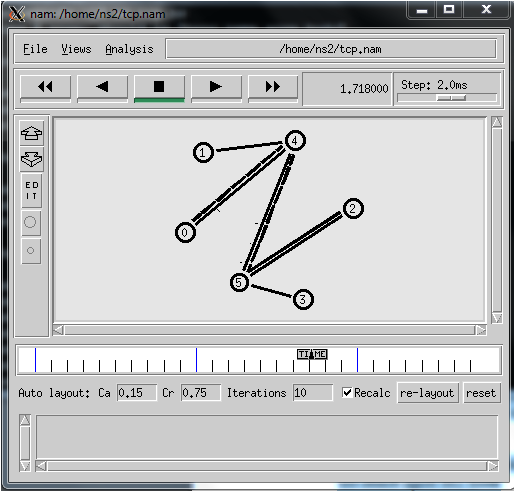
$ns at 0.0 "$ftp start"

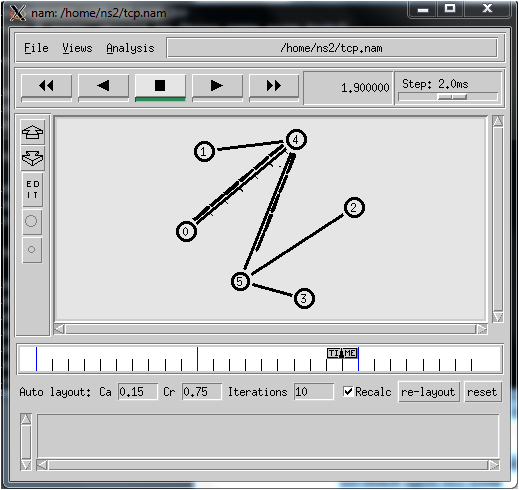
$ns at 2.5 "$ftp stop"

$ns at 3 "finish"

$ns run

**OUTPUT:**





**RESULT:**

Thus the program for implementing TCP was executed using NS-2 and output verified using Network Animator.

**Ex No: 3a DISTANCE VECTOR ROUTING PROTOCOL**

**Date :**

**AIM:**

To simulate a link failure and to observe distance vector routing protocol in action.

**ALGORITHM:**

1. Create a simulator object

2. Set routing protocol to Distance Vector routing

3. Trace packets on all links onto NAM trace and text trace file

4. Define finish procedure to close files, flush tracing and run NAM

5. Create eight nodes

6. Specify the link characteristics between nodes

7. Describe their layout topology as a octagon

8. Add UDP agent for node n1

9. Create CBR traffic on top of UDP and set traffic parameters.

10. Add a sink agent to node n4

11. Connect source and the sink

12. Schedule events as follows:

a. Start traffic flow at 0.5

b. Down the link n3-n4 at 1.0

c. Up the link n3-n4 at 2.0

d. Stop traffic at 3.0

e. Call finish procedure at 5.0

13. Start the scheduler

14. Observe the traffic route when link is up and down

15. View the simulated events and trace file analyze it

16. Stop the program.

**PROGRAM:**

set ns [new Simulator]

$ns rtproto DV

set nf [open out.nam w]

$ns namtrace-all $nf

set nt [open trace.tr w]

$ns trace-all $nt

proc finish {} {

global ns nf

$ns flush-trace

close $nf

exec nam -a out.nam &

exit 0

}

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]

set n8 [$ns node]

$ns duplex-link $n1 $n2 1Mb 10ms DropTail

$ns duplex-link $n2 $n3 1Mb 10ms DropTail

$ns duplex-link $n3 $n4 1Mb 10ms DropTail

$ns duplex-link $n4 $n5 1Mb 10ms DropTail

$ns duplex-link $n5 $n6 1Mb 10ms DropTail

$ns duplex-link $n6 $n7 1Mb 10ms DropTail

$ns duplex-link $n7 $n8 1Mb 10ms DropTail

$ns duplex-link $n8 $n1 1Mb 10ms DropTail

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

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

$ns duplex-link-op $n3 $n4 orient right-up

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

$ns duplex-link-op $n5 $n6 orient right-down

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

$ns duplex-link-op $n7 $n8 orient left-down

$ns duplex-link-op $n8 $n1 orient left

set udp0 [new Agent/UDP]

$ns attach-agent $n1 $udp0

set cbr0 [new Application/Traffic/CBR]

$cbr0 set packetSize\_ 500

$cbr0 set interval\_ 0.005

$cbr0 attach-agent $udp0

set null0 [new Agent/Null]

$ns attach-agent $n4 $null0

$ns connect $udp0 $null0

$ns at 0.0 "$n1 label Source"

$ns at 0.0 "$n4 label Destination"

$ns at 0.5 "$cbr0 start"

$ns rtmodel-at 1.0 down $n3 $n4

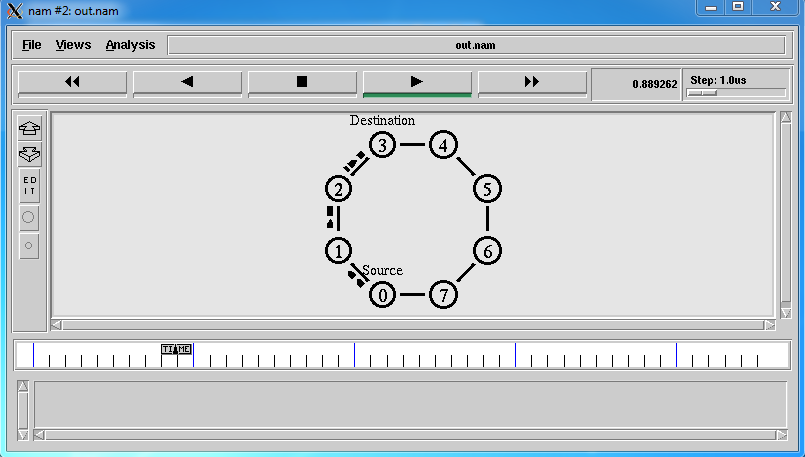
$ns rtmodel-at 2.0 up $n3 $n4

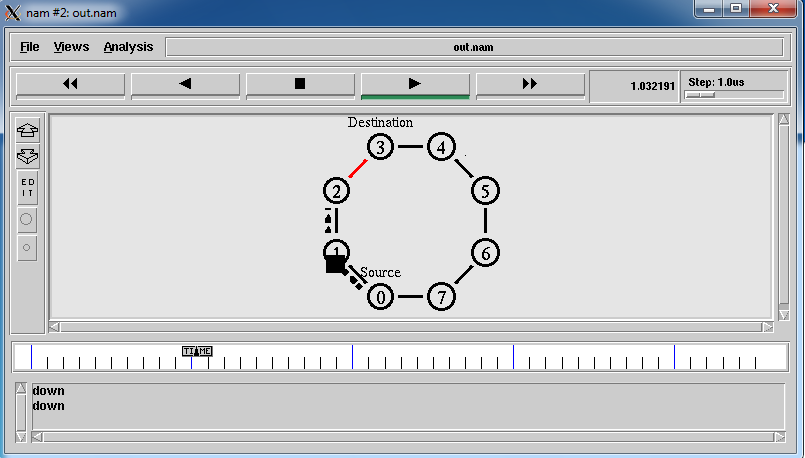
$ns at 4.5 "$cbr0 stop"

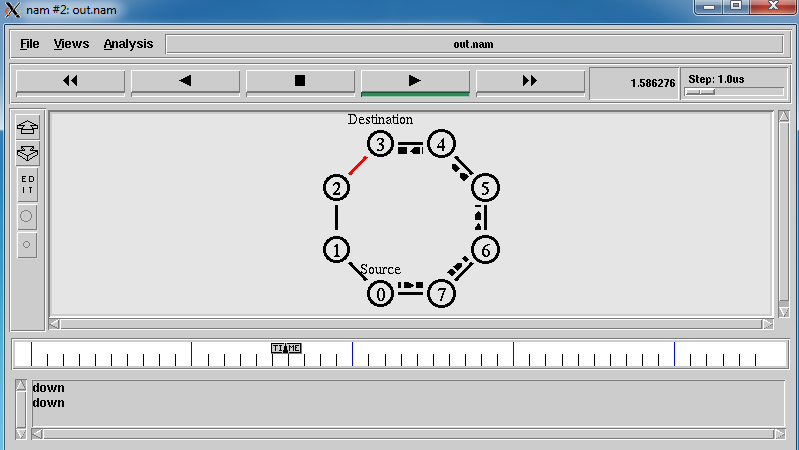
$ns at 5.0 "finish"

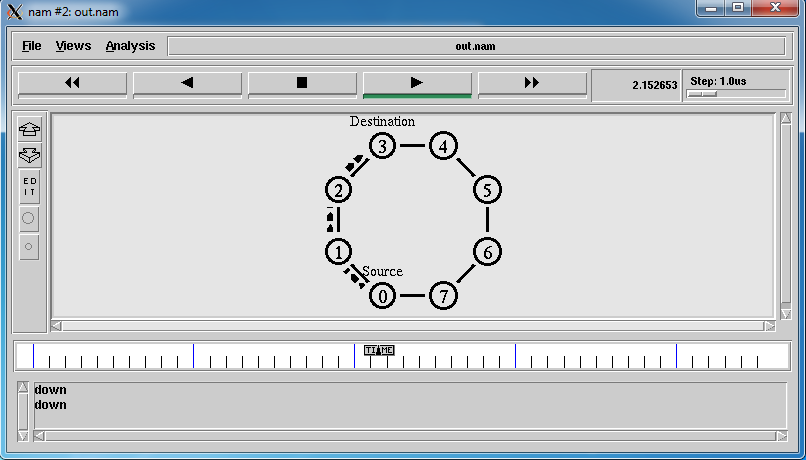
$ns run

**OUTPUT:**









**RESULT:**

Thus, performance of distance vector protocol and routing path was studied using NS2.

**Ex No: 3b LINK STATE ROUTING PROTOCOL**

**Date :**

**AIM:**

To simulate a link failure and to observe link state routing protocol in action.

**ALGORITHM:**

1. Create a simulator object

2. Set routing protocol to link state routing

3. Trace packets on all links onto NAM trace and text trace file

4. Define finish procedure to close files, flush tracing and run NAM

5. Create four nodes

6. Specify the link characteristics between nodes

7. Describe their layout topology as a quad node.

8. Add TCP agent for node n0

9. Create FTP traffic on top of TCP and set traffic parameters.

10. Add a sink agent to node n3

11. Add UDP agent for node n2

12. Create CBR traffic on top of UDP and set traffic parameters.

13. Connect source and the sink

14. Schedule events as follows:

a. Start traffic flow at 0.0

b. Down the link n1-n3 at 1.0

c. Up the link n1-n3 at 2.0

d. Call finish procedure at 5.0

15. Start the scheduler

16. Observe the traffic route when link is up and down

17. View the simulated events and trace file analyze it

18. Stop

**PROGRAM :**

set ns [new Simulator]

set nf [open out.nam w]

$ns namtrace-all $nf

set tr [open out.tr w]

$ns trace-all $tr

proc finish {} {

global nf ns tr

$ns flush-trace

close $tr

exec nam out.nam &

exit 0

}

set n0 [$ns node]

set n1 [$ns node]

set n2 [$ns node]

set n3 [$ns node]

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

$ns duplex-link $n1 $n3 10Mb 10ms DropTail

$ns duplex-link $n2 $n1 10Mb 10ms DropTail

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

$ns duplex-link-op $n1 $n3 orient right

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

set tcp [new Agent/TCP]

$ns attach-agent $n0 $tcp

set ftp [new Application/FTP]

$ftp attach-agent $tcp

set sink [new Agent/TCPSink]

$ns attach-agent $n3 $sink

set udp [new Agent/UDP]

$ns attach-agent $n2 $udp

set cbr [new Application/Traffic/CBR]

$cbr attach-agent $udp

set null [new Agent/Null]

$ns attach-agent $n3 $null

$ns connect $tcp $sink

$ns connect $udp $null

$ns rtmodel-at 1.0 down $n1 $n3

$ns rtmodel-at 2.0 up $n1 $n3

$ns rtproto LS

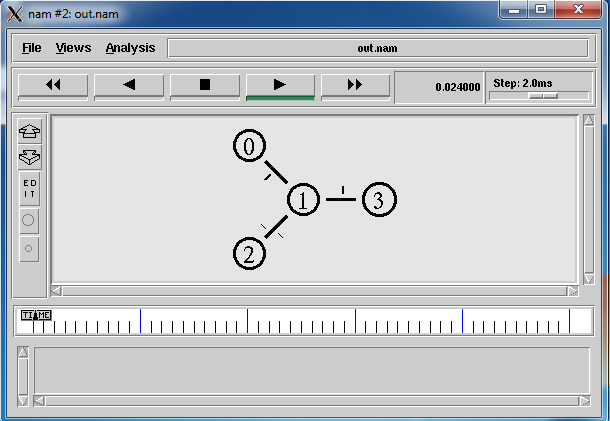
$ns at 0.0 "$ftp start"

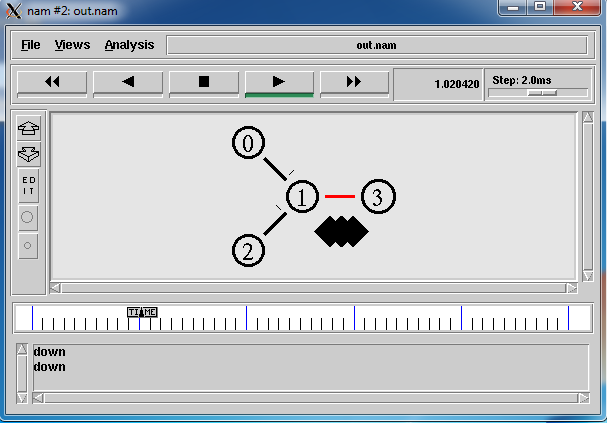
$ns at 0.0 "$cbr start"

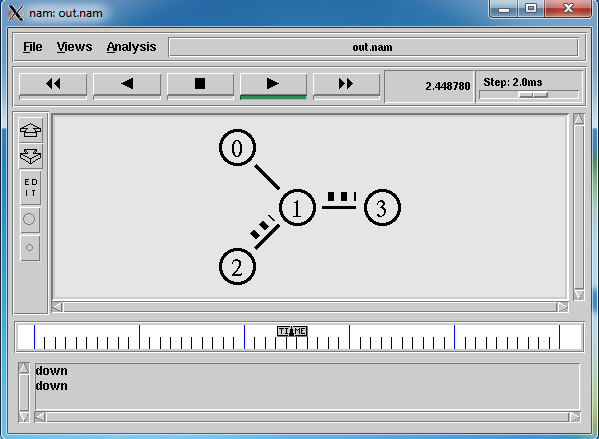
$ns at 5.0 "finish"

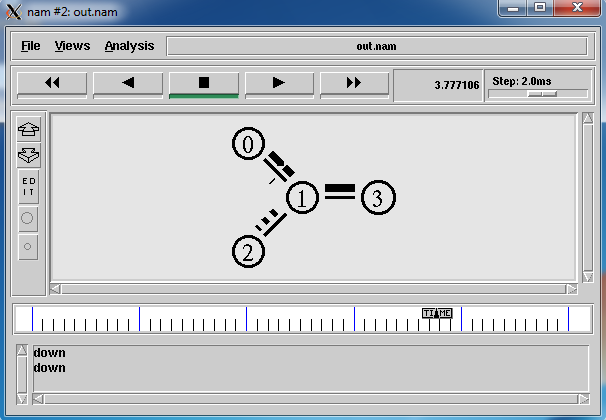
$ns run

**OUTPUT:**









**RESULT:**

Thus, performance of link state protocol and routing path was studied using NS2.

**EX:NO:5 PERFORMANCE COMPARISON OF ROUTING PROTOCOLS**

**Date :**

**AIM:**

To Study of performance evaluation of routing protocols using simulation tool.

**ALGORITHM:**

Step1: Create a simulator object.

Step2: Open a out trace file.

Step3: Open NAM trace file.

Step4: Specify link characteristic.

Step5: Specify layout as the octagon.

Step6: Create a UDP agent and attach it to load N1

Step7: Create a CBR traffic source and attach it to udp0

Step8: Create a Null agent (a traffic sink) and attach it to node n4

Step9: Connect the traffic source with the traffic sink

Step10: Schedule events for the CBR agent and the network dynamics

Step11: Run the simulation

**PROGRAM:**

set ns [new Simulator]

$ns multicast

set f [open out.tr w]

$ns trace-all $f

$ns namtrace-all [open out.nam w]

$ns color 1 red

# prune/graft packets

$ns color 30 purple

$ns color 31 green set n0 [$ns node] set n1 [$ns node] set n2 [$ns node] set n3 [$ns node]

# Use automatic layout

$ns duplex-link $n0 $n1 1.5Mb 10ms DropTail

$ns duplex-link $n1 $n2 1.5Mb 10ms DropTail

$ns duplex-link $n1 $n3 1.5Mb 10ms DropTail

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

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

$ns duplex-link-op $n1 $n3 orient right-down

$ns duplex-link-op $n0 $n1 queuePos 0.5 set mrthandle [$ns mrtproto DM {}]

set cbr0 [new Application/Traffic/CBR] set udp0 [new Agent/UDP]

$cbr0 attach-agent $udp0

$ns attach-agent $n1 $udp0

$udp0 set dst\_ 0x8001

set cbr1 [new Application/Traffic/CBR] set udp1 [new Agent/UDP]

$cbr1 attach-agent $udp1

$udp1 set dst\_ 0x8002

$udp1 set class\_ 1

$ns attach-agent $n3 $udp1

set rcvr [new Agent/LossMonitor] #$ns attach-agent $n3 $rcvr

$ns at 1.2 "$n2 join-group $rcvr 0x8002"

$ns at 1.25 "$n2 leave-group $rcvr 0x8002"

$ns at 1.3 "$n2 join-group $rcvr 0x8002"

$ns at 1.35 "$n2 join-group $rcvr 0x8001"

$ns at 1.0 "$cbr0 start"

$ns at 1.1 "$cbr1 start"

$ns at 2.0 "finish" proc finish {} {

global ns

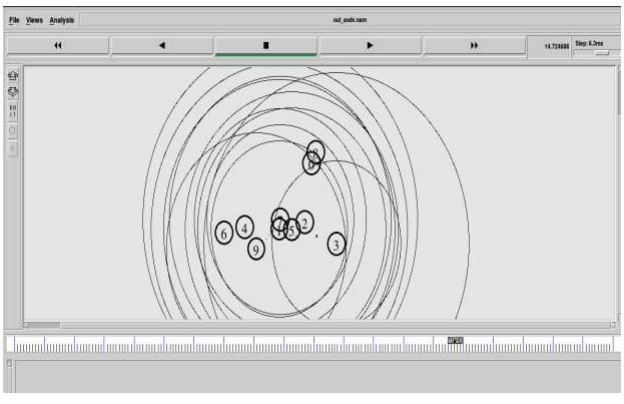
$ns flush-trace

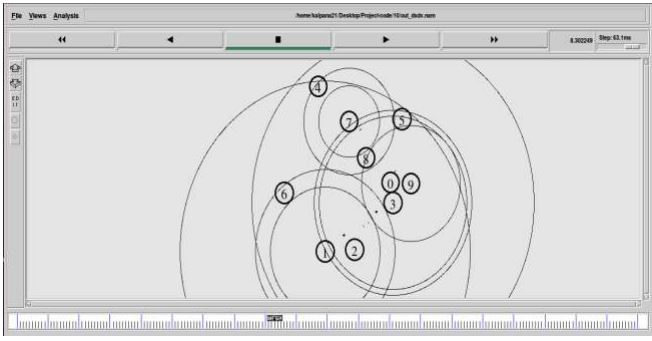
puts "running nam..." exec nam out.nam & exit 0

}

$ns run

**OUTPUT:**





**RESULT:**

Thus, performance of comparison of routing protocols was studied using NS2.