

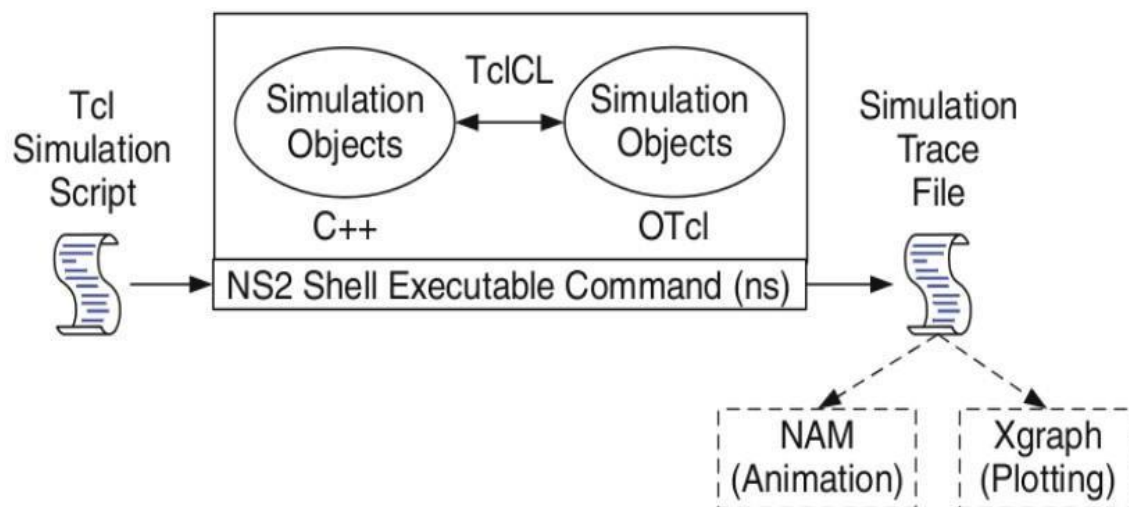
EX.NO:7	Study of Network simulator (NS) and Simulation of Congestion Control Algorithms using NS
DATE:	

AIM:

To Study Network simulator (NS) and to Simulate the Congestion Control Algorithms using NS2.

STUDY OF NS:**THEORY:**

Network Simulator (Version 2), widely known as NS2, is simply an event driven simulation tool that has proved useful in studying the dynamic nature of communication networks. Simulation of wired as well as wireless network functions and protocols (e.g., routing algorithms, TCP, UDP) can be done using NS2. In general, NS2 provides users with a way of specifying such network protocols and simulating their corresponding behaviors. Due to its flexibility and modular nature, NS2 has gained constant popularity in the networking research community since its birth in 1989.

Basic Architecture of NS2:

The above figure shows the basic architecture of NS2. NS2 provides users with an executable command `ns` which takes on input argument, the name of a Tcl simulation scripting file. Users are feeding the name of a Tcl simulation script (which sets up a simulation) as an input argument of an NS2 executable command `ns`.

In most cases, a simulation trace file is created, and is used to plot graph and/or to create animation. NS2 consists of two key languages: C++ and Object-oriented Tool Command Language(OTCL).

While the C++ defines the internal mechanism (i.e., a backend) of the simulation objects, the OTcl sets up simulation by assembling and configuring the objects as well as scheduling discrete events (i.e., a frontend).

The C++ and the OTcl are linked together using TclCL. Mapped to a C++ object, variables in the OTcl domains are sometimes referred to as handles. Conceptually, a handle (e.g., `n` as a Node handle) is just a string (e.g., `o10`) in the OTcl domain, and does not contain any functionality. Instead, the functionality

(e.g., receiving a packet) is defined in the mapped C++ object (e.g., of class `Connector`). In the OTcl domain, a handle acts as a frontend which interacts with users and other OTcl objects. It may defines its

own procedures and variables to facilitate the interaction. Note that the member procedures and variables in the OTcl domain are called instance procedures (instprocs) and instance variables (instvars), respectively

Tcl scripting:

- Tcl is a general purpose scripting language. [Interpreter]
- Tcl runs on most of the platforms such as Unix, Windows, and Mac.
- The strength of Tcl is its simplicity.
- It is not necessary to declare a data type for variable prior to the usage

Basics of TCL :

Syntax: command arg1 arg2 arg3 **Hello World!**

```
puts stdout{Hello, World!}
```

Hello, World!

Variables :

Command Substitution

```
set a 5
```

```
set len [string length foobar]
```

```
set b $a
```

```
set len [expr [string length foobar] + 9]
```

Simple Arithmetic:

```
proc Diag {a b} {
```

```
set c [expr sqrt($a * $a + $b * $b)]
```

```
return $c }
```

```
puts —Diagonal of a 3, 4 right triangle is [Diag 3 4]
```

Output: Diagonal of a 3, 4 right triangle is 5.0

Loops

```
while{$i < $n} {
```

```
...
```

```
}
```

```
for {set i 0} {$i < $n} {incr i}
```

```
{
```

```
...
```

```
}
```

NS Simulator Preliminaries :

1. Initialization and termination aspects of the ns simulator.
2. Definition of network nodes, links, queues and topology.
3. Definition of agents and of applications.
4. The nam visualization tool.
5. Tracing and random variables.

Initialization and Termination of TCL Script in NS-2 :

An ns simulation starts with the command

```
set ns [new Simulator]
```

Which is thus the first line in the tcl script? This line declares a new variable as using the set command, you can call this variable as you wish, In general people declares it as ns because it is an instance of the Simulator class, so an object the code[new Simulator] is indeed the installation of the class Simulator using the reserved word new.

In order to have output files with data on the simulation (trace files) or files used for visualization (nam files), we need to create the files using —"open" command:

```
#Open the Trace file
set tracefile1 [open out.trw]
$ns trace-all $tracefile1
#Open the NAM trace file
set namfile [open out.nam w]
$ns namtrace-all $namfile
```

The above creates a dta trace file called —out.trl and a nam visualization trace file called —out nam. Within the tcl script, these files are not called explicitly by their names, but instead by pointers that are declared above and called —tracefile1 and —nam file respectively. Remark that they begins with a # symbol. The second line open the file —out.trl to be used for writing, declared with the letter —w. The third line uses a simulator method called trace-all that have as parameter the name of the file where the traces will go.

The last line tells the simulator to record all simulation traces in NAM input format. It also gives the file name that the trace will be written to later by the command \$ns flush-trace.

In our case, this will be the file pointed at by the pointer —\$namfile ,i.e the file —out.trl.

The termination of the program is done using a —finish procedure.

```
#Define a 'finish' procedure
Proc finish { } {
    global ns tracefile1 namfile
    $ns flush-trace
    Close $tracefile1
    Close $namfile
    Exec nam out.nam &
    Exit 0
```

The word proc declares a procedure in this case called **finish** and without arguments. The word **global** is used to tell that we are using variables declared outside the procedure. The simulator method —**flush-trace** will dump the traces on the respective files. The tcl command —**close** closes the trace files defined before and **exec** executes the nam program for visualization.

The command **exit** will ends the application and return the number 0 as status to the system. Zero is the default for a clean exit. Other values can be used to say that is a exit because something fails.

At the end of ns program we should call the procedure —**finish** and specify at what time the termination should occur. For example,

\$ns at 125.0 "finish"

will be used to call —**finish** at time 125sec. Indeed, the **at** method of the simulator allows us to schedule events explicitly.

The simulation can then begin using the command

\$ns run

Definition of a network of links and nodes :

The way to define a node is

```
set n0 [$ns node]
```

The node is created which is printed by the variable n0. When we shall refer to that node in the script we shall thus write \$n0.

Once we define several nodes, we can define the links that connect them. An example of a definition of a link is:

\$ns duplex-link \$n0 \$n2 10Mb 10ms DropTail

Which means that \$n0 and \$n2 are connected using a bi-directional link that has 10ms of propagation delay and a capacity of 10Mb per sec for each direction. To define a directional link instead of a bi-directional

one, we should replace “duplexlink” by “simplexlink”.

In NS, an output queue of a node is implemented as a part of each link whose input is that node. The definition of the link then includes the way to handle overflow at that queue. In our case, if the buffer capacity of the output queue is exceeded then the last packet to arrive is dropped. Many alternative options exist, such as the RED (Random Early Discard) mechanism, the FQ (Fair Queuing), the DRR (Deficit Round Robin), the stochastic Fair Queuing (SFQ) and the CBQ (which including a priority and a round-robin scheduler). In ns, an output queue of a node is implemented as a part of each link whose input is that node. We should also define the buffer capacity of the queue related to each link. An example would be:

#set Queue Size of link (n0-n2) to 20

\$ns queue-limit \$n0 \$n2 20

Agents and Applications:

We need to define routing (sources, destinations) the agents (protocols) the application that use them FTP over TCP

TCP is a dynamic reliable congestion control protocol. It uses Acknowledgements created by the destination to know whether packets are well received. There are number variants of the TCP protocol, such as Tahoe, Reno, NewReno, Vegas. The type of agent appears in the first line:

set tcp [new Agent/TCP]

The command **\$ns attach-agent \$n0 \$tcp** defines the source node of the tcp connection. The command

set sink [new Agent /TCPSink]

Defines the behavior of the destination node of TCP and assigns to it a pointer called sink

#Setup a UDP connection

```
set udp [new Agent/UDP]
$ns attach-agent
  $n1
  $udp
set null [new Agent/Null]
$ns attach-agent $n5 $null
$ns connect $udp $null
$udp set fid_2
```

#setup a CBR over UDP connection

```
set cbr [new
Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set packetize_ 100
$cbr set rate_ 0.01Mb
$cbr set random_ false
```

Above shows the definition of a CBR application using a UDP agent. The command **\$ns attach-agent \$n4 \$sink** defines the destination node. The command **\$ns connect \$tcp \$sink** finally makes the TCP connection between the source and destination nodes.

TCP has many parameters with initial fixed defaults values that can be changed if mentioned explicitly. For example, the default TCP packet size has a size of 1000bytes. This can be changed to another value,

say 552bytes, using the command `$tcp set packetSize_ 552`. When we have several flows, we may wish to distinguish them so that we can identify them with different colors in the visualization part. This is done by the command `$tcp set fid_ 1` that assigns to the TCP connection a flow identification of “1”. We shall later give the flow identification of “2” to the UDP connection.

CBR over UDP :

A UDP source and destination is defined in a similar way as in the case of TCP. Instead of defining the rate in the command `$cbr set rate_ 0.01Mb`, one can define the time interval between transmission of packets using the command.

\$cbr set interval_ 0.005

The packet size can be set to some value using

\$cbr set packetSize_ <packet size>

Scheduling Events :

NS is a discrete event based simulation. The tcp script defines when event should occur. The initializing command `set ns [new Simulator]` creates an event scheduler, and events are then scheduled using the format:

\$ns at <time> <event>

The scheduler is started when running ns that is through the command `$ns run`. The beginning and end of the FTP and CBR application can be done through the following command

\$ns at 0.1 “\$cbr start” \$ns at 1.0 “\$ftp start”

\$ns at 124.0 “\$ftp stop”

\$ns at 124.5 “\$cbr stop”

CONGESTION CONTROL ALGORITHM:

ALGORITHM:

Step 1: Start the program

Step 2: Create a simulation object

Step 3: Set routing protocol to routing

Step 4: Trace packets and all links onto NAM trace and to trace file

Step 5: Create right nodes

Step 6: Describe their layout topology as octogan

Step 7: Add a sink agent to node

Step 8: Connect source and sink

Step 9: Stop the program

PROGRAM:

```
set ns [new Simulator]
```

```
set nr [open thor.red.tr w]
```

```
$ns trace-all $nr
```

```
set nf [open thro.nam w]
```

```
$ns namtrace-all $nf
```

```
proc finish { } {
```

```
global ns nr nf
```

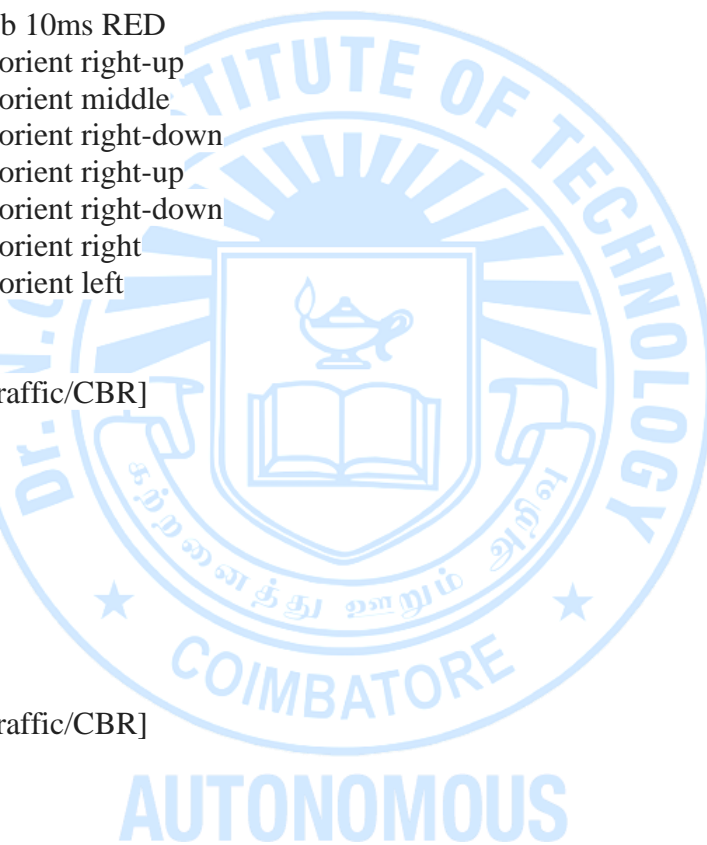
```
$ns flush-trace
```

```
close $nf
```

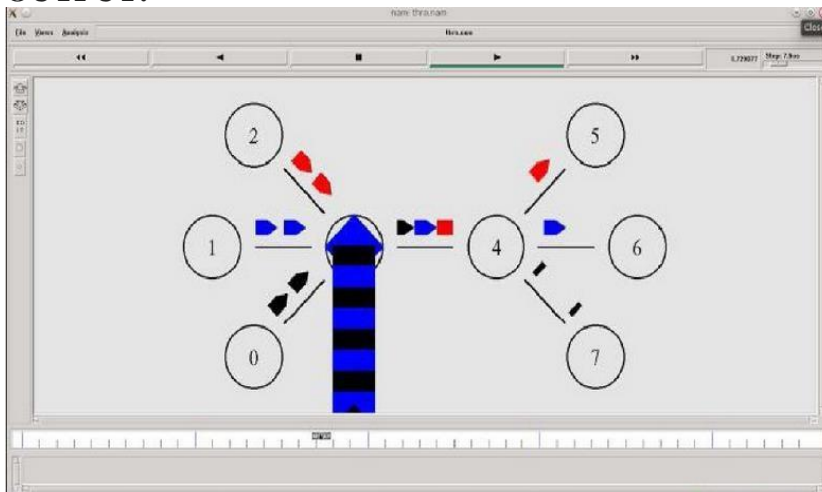
```
close $nr
```



```
    exec nam thro.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]
set n6 [$ns node]
set n7 [$ns node]
$ns duplex-link $n0 $n3 1Mb 10ms RED
$ns duplex-link $n1 $n3 1Mb 10ms RED
$ns duplex-link $n2 $n3 1Mb 10ms RED
$ns duplex-link $n3 $n4 1Mb 10ms RED
$ns duplex-link $n4 $n5 1Mb 10ms RED
$ns duplex-link $n4 $n6 1Mb 10ms RED
$ns duplex-link $n4 $n7 1Mb 10ms RED
$ns duplex-link-op $n0 $n3 orient right-up
$ns duplex-link-op $n3 $n4 orient middle
$ns duplex-link-op $n2 $n3 orient right-down
$ns duplex-link-op $n4 $n5 orient right-up
$ns duplex-link-op $n4 $n7 orient right-down
$ns duplex-link-op $n1 $n3 orient right
$ns duplex-link-op $n6 $n4 orient left
set udp0 [new Agent/UDP]
$ns attach-agent $n2 $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 $n5 $null0
$ns connect $udp0 $null0
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 attach-agent $udp1
set null0 [new Agent/Null]
$ns attach-agent $n6 $null0
$ns connect $udp1 $null0
set udp2 [new Agent/UDP]
$ns attach-agent $n0 $udp2
set cbr2 [new Application/Traffic/CBR]
$cbr2 set packetSize_ 500
$cbr2 set interval_ 0.005
$cbr2 attach-agent $udp2
set null0 [new Agent/Null]
$ns attach-agent $n7 $null0
$ns connect $udp2 $null0
$udp0 set fid_ 1
$udp1 set fid_ 2
$udp2 set fid_ 3
$ns color 1 Red
$ns color 2 Green
```



```
$ns color 2 blue
$ns at 0.1 "$cbr0 start"
$ns at 0.2 "$cbr1 start"
$ns at 0.5 "$cbr2 start"
$ns at 4.0 "$cbr2 stop"
$ns at 4.2 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"
$ns at 5.0 "finish"
$ns run
```

OUTPUT:**RESULT:**

Thus the study of Network Simulator (NS) and the study of stimulating Congestion Control algorithms using NS2 is verified successfully.