AIM:

To study about NS2 simulator in detail.

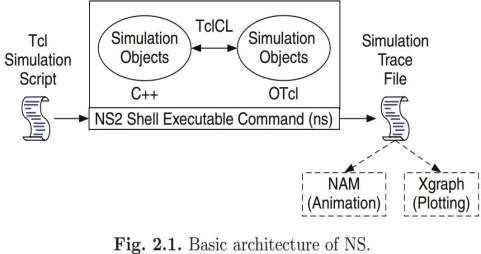
# THEORY:

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. Ever since, several revolutions and revisions have marked the growing maturity of the tool, thanks to substantial contributions from the players in the field. Among these are the University of California and Cornell University who developed the REAL network simulator,1 the foundation which NS is based on. Since 1995 the Defense Advanced Research Projects Agency (DARPA) supported development of NS through the Virtual Inter Network Testbed (VINT) project . Currently the National Science Foundation (NSF) has joined the ride in development. Last but not the least, the group of Researchers and developers in the community are constantly working to keep NS2 strong and versatile.

Figure 2.1 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).

# BASIC ARCHITECTURE:

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. Before proceeding further, the readers are encouraged to learn C++ and OTcl languages. We refer the readers to [14] for the detail of C++, while a brief tutorial of Tcl and OTcl tutorial are given in Appendices A.1 and A.2, respectively.

NS2 provides a large number of built-in C++ objects. It is advisable to use these C++ objects to set up a simulation using a Tcl simulation script. However, advance users may find these objects insufficient. They need to develop their own C++ objects, and use a OTcl configuration interface to put together these objects. After simulation, NS2 outputs either text-based or animation-based simulation results. To interpret these

results graphically and interactively, tools such as NAM (Network AniMator) and XGraph are used. To analyze a particular behaviour of the network, users can extract a relevant subset of text-based data and transform it to a more conceivable presentation.

# CONCEPT OVERVIEW:

NS uses two languages because simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols requires a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important. On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios.

In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important. ns meets both of these needs with two languages, C++ and OTcl.

1. 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.

1. Basics of TCL Syntax: command arg1 arg2 arg3
2. 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]

1. Wired TCL Script Components Create the event scheduler

Open new files &

turn on the tracing Create the nodes Setup the links

Configure the traffic type (e.g., TCP, UDP, etc) Set the time of traffic generation (e.g., CBR, FTP) Terminate the simulation

1. NS Simulator Preliminaries.
2. Initialization and termination aspects of the ns simulator.
3. Definition of network nodes, links, queues and topology.
4. Definition of agents and of applications.
5. The nam visualization tool.
6. Tracing and random variables.
7. Initialization and Termination of TCL Script in NS-2 An ns simulation starts with the command
8. 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.tr w]**

**$ns trace-all $tracefile #Open the NAM trace file set namfile [open out.nam w]**

**$ns namtrace-all $namfile**

The above creates a dta trace file called out.tr 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 ―namfile respectively. Remark that they begins with a # symbol. The second line open the file ―out.tr 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.

**Define a “finish‟ procedure Proc finish { } {**

**global ns tracefile1 namfile**

**$ns flush-trace Close $tracefile1 Close $namfile**

**Exec nam out.nam & Exit 0**

**}**

**Definition of a network of links and nodes**

The way to define a node is

1. set n0 [$ns node]

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

1. $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 ―duplex-link by

―simplex-link.

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**

**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:

1. 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**

**The below 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**

1. $tcp $sink finally makes the TCP connection between the source and destination nodes.

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

**$cbr attach-agent $udp**

**$cbr set packetsize\_ 100**

**$cbr set rate\_ 0.01Mb**

**$cbr set random\_ false**

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.

# RESULT:

Thus the network simulator 2 is studied in detail.