CHAPTER 7

NETWORK SIMULATOR

7.1 INTRODUCTION

The Network Simulator (NS2) is an object oriented simulator, written in C++, with an Object oriented Tool command language (OTcl) interpreter as a front-end. The simulator supports a class hierarchy in C++ and a similar class hierarchy within the OTcl interpreter. The two hierarchies are closely related to each other. From the user’s perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy.

The root of this hierarchy is the class Tcl Object. Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class Tcl Class. User instantiated objects are mirrored through methods defined in the class Tcl Object.

The NS components are mostly compound network components. Figure 7.1 shows a partial OTcl class hierarchy of NS, which help in understanding the basic network components.
NS is an event driven network simulator developed at UC Berkeley that simulates a variety of IP networks. It implements network protocols such as TCP and UDP, traffic source behaviors 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 more. NS also implements multicasting and some of the MAC layer protocols for LAN simulations.

![Class hierarchies of TCL (Partial)](image)

**Figure 7.1 Class hierarchies of TCL (Partial)**

The NS project is now a part of the VINT project that develops tools for simulation results display, analysis and converters that convert network topologies generated by well-known generators to NS formats. Currently, NS (version 2) written in C++ and OTcl (Tcl script language with Object-oriented extensions developed at MIT) is available.
7.1.1 Simplified User's View of Network simulator

![Diagram of the simplified view of the network simulator]

**Figure 7.2 View of network simulator**

As shown in Figure 7.2, in a simplified user's view, NS is an Object-oriented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component object libraries, and network set up (plumbing) module libraries. Plumbing modules are implemented as member functions of the base simulator object.

In other words, to use NS, the program is written using OTcl script language. To set up and run a simulation network, a user should write an OTcl script that initiates an event scheduler. It sets up the network topology using the network objects and the plumbing functions in the library, and tells traffic sources when to start and stop transmitting packets through the event scheduler.

The term 'plumbing' is used for a network setup, because setting up a network is plumbing possible data paths among network objects by setting the ‘neighbour’ pointer of an object to the address of an appropriate object.

When a user wants to make a new network object, he or she can easily make an object either by writing a new object or by making a compound object from the object library, and plumb the data path through the
object. This may sound like complicated job, but the plumbing OTcl modules actually make the job very easy. The power of NS comes from this plumbing.

Another major component of NS beside network objects is the event scheduler. An event in NS is a packet ID that is unique for a packet with scheduled time and the pointer to an object that handles the event. In NS, an event scheduler keeps track of simulation time and fires all the events in the event queue scheduled for the current time by invoking appropriate network components.

These usually are the ones which issued the events, and let them do the appropriate action associated with packet pointed by the event. Network components communicate with one another passing packet. However this does not consume actual simulation time. All the network components that need to spend some simulation time handling a packet (i.e. need a delay) use the event scheduler by issuing an event for the packet and waiting for the event to be fired to itself before doing further action handling the packet.

For example, a network switch component that simulates a switch with 20 microseconds of switching delay issues an event for a packet to be switched to the scheduler as an event 20 microsecond later. The scheduler after 20 microseconds de queues the event and fires it to the switch component, which then passes the packet to an appropriate output link component. Another use of an event scheduler is timer.

For example, TCP needs a timer to keep track of a packet transmission time out for retransmission (transmission of a packet with the same TCP packet number but different NS packet ID). Timers use event schedulers in a similar manner. The only difference is that timer measures a
time value associated with a packet and does an appropriate action related to that packet after a certain time goes by and it does not simulate a delay.

NS is written not only in OTcl but in C++ also. For efficiency reason, NS separates the data path implementation from control path implementations. In order to reduce packet and event processing time (not simulation time), the event scheduler and the basic network component objects in the data path are written and compiled using C++.

These compiled objects are made available to the OTcl interpreter through an OTcl linkage that creates a matching OTcl object for each of the C++ objects and makes the control functions and the configurable variables specified by the C++ object act as member functions and member variables of the corresponding OTcl object. In this way, the controls of the C++ objects are given to OTcl. It is also possible to add member functions and variables to a C++ linked OTcl object.

The objects in C++ that do not need to be controlled in a simulation or internally used by another object do not need to be linked to OTcl. Likewise, an object (not in the data path) can be entirely implemented in OTcl. Figure 7.1 shows an object hierarchy example in C++ and OTcl. One thing to note in the figure 7.3 is that for C++ objects that have an OTcl linkage forming a hierarchy, there is a matching OTcl object hierarchy very similar to that of C++.
Figure 7.3 C++ and OTcl: The Duality

Figure 7.4 Architectural View of Network Simulator

Figure 7.4 shows the general architecture of NS. In this figure a general user (not an NS developer) can be thought of standing at the left bottom corner, designing and running simulations in Tcl using the simulator objects in the OTcl library. The event schedulers and most of the network components are implemented in C++ and available to OTcl through an OTcl linkage that is implemented using Tcl. The whole thing together makes NS,
which is a Object Oriented extended Tcl interpreter with network simulator libraries.

As shown in Figure 7.2, when a simulation is finished, NS produces one or more text-based output files that contain detailed simulation data, if specified to do so in the input Tcl (or more specifically, OTcl) script. The data can be used for simulation analysis (examples of two simulation result analyse examples are presented in later sections) or as an input to a graphical simulation display tool called Network Animator (NAM) that is developed as a part of VINT project.

NAM has a nice graphical user interface similar to that of a CD player (play, fast forward, rewind, pause and so on), and also has a display speed controller. Furthermore, it can graphically present information such as throughput and number of packet drops at each link, although the graphical information cannot be used for accurate simulation analysis.

In general, an NS script starts with making a simulator object instance.

- Set \texttt{ns} [new Simulator] generates an NS simulator object instance, and assigns it to variable \texttt{ns}
- Initialize the packet format
- Create a scheduler (default is calendar scheduler)
- Select the default address format

The ‘Simulator’ object has member functions that do the following:

- Create compound objects such as nodes and links
- Connect network component objects created (ex. attach-agent)
- Set network component parameters (mostly for compound objects)
- Create connections between agents (e.g. make connection between a ‘tcp’ and ‘sink’)
- Specify NAM display options

7.2 EVENT SCHEDULER

As described in the Overview section, the main users of an event scheduler are network components that simulate packet-handling delay or that need timers. Figure 7.5 shows each network object using an event scheduler. Note that a network object that issues an event is the one which handles the event later at scheduled time. Also note that the data path between network objects is different from the event path.

![Figure 7.5 Discrete Event Scheduler](image)

NS has two different types of event scheduler implemented. These are real-time and non-real-time schedulers. For a non-real-time scheduler, three implementations (List, Heap and Calendar) are available even though
they all logically perform the same. This is because of backward compatibility: some early implementation of network components added by a user (not the original ones included in a package) may use a specific type of scheduler not through public functions but by hacking around the internals.

The Calendar non-real-time scheduler is set as the default. The real-time scheduler is for emulation, which allows the simulator to interact with a real network. Currently, emulation is under development although an experimental version is available.

### 7.2.1 Tracing

In NS, network activities are traced around simplex links. If the simulator is directed to trace network activities (specified using $\texttt{ns trace-all file}$ or $\texttt{ns namtrace-all file}$), the links created after the command will have the following trace objects inserted as shown in Figure 5.6. Users can also specifically create a trace object of type between the given source and destination nodes using the create-trace \{type file source st\} command.

When each inserted trace object (i.e. EnqT, DeqT, DrpT and RecvT) receives a packet, it writes to the specified trace file without consuming any simulation time, and passes the packet to the next network object. The trace format will be examined in the General Analysis Example section.

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
<th>From node</th>
<th>To node</th>
<th>Pkt type</th>
<th>Pkt size</th>
<th>Flags</th>
<th>Fid</th>
<th>Src addr</th>
<th>Dst addr</th>
<th>Seq num</th>
<th>Pkt id</th>
</tr>
</thead>
</table>
r : receive (at to_node)
+ : enqueue (at queue)  src_addr : node.port [3.0]
d : drop   (at queue)
d

```
  r 1.3556 3 2 ack 40 ------- 1 3.0 0.0 15 201
  + 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
- 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
  r 1.35576 0 2 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
  d 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
- 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
```

**Figure 7.6 Packets and Trace File Format**

Having simulation trace data at hand, all one has to do is to transform a subset of the data of interest into a comprehensible information and analyze it. Down below is a small data transformation example. This example uses a command written in perl called ‘column’ that selects columns of given input. Following is a tunneled shell command combined with awk, which calculates CBR traffic jitter at receiver node (n3) using data in ‘out.tr’, and stores the resulting data in ‘jitter.txt’.

### 7.2.2 Packet

A NS packet is composed of a stack of headers, and an optional data space. As briefly mentioned in the "Simple Simulation Example" section, a packet header format is initialized when a simulator object is created, where a stack of all registered (or possibly useable) headers, such as the common header that is commonly used by any objects as needed, IP header, TCP header, RTP header (UDP uses RTP header) and trace header, is defined and the offset of each header in the stack is recorded. This means that whether or
not a specific header is used, a stack composed of all registered headers is created when a packet is allocated by an agent. A network object can access any header in the stack of a packet it processes using the corresponding offset value.

![NS Packet Format](image)

**Figure 7.7 NS Packet Format**

Usually, a packet only has the header stack (and a data space pointer that is null). Although a packet can carry actual data (from an application) by allocating a data space, very few application and agent implementations support this. This is because it is meaningless to carry data around in a non-real-time simulation. Another possible approach would be creating a new header for the application and modifying the underlying agent to write the data received from the application to the new header.
7.3 NAM SUPPORT

NAM is a Tcl based animation tool for viewing network simulation traces and real world packet trance data. The design theory behind nam was to create an animator that is able to read large animation data sets and be extensible enough so that it could be used in different network visualization situations.

The first step to use nam is to produce the trace file. The trace file contains topology information. Usually the trace file is generated by ns. During an ns simulation, the user can produce topology configurations, lay out information and packet traces using tracing events in ns.

Figure 7.8 NAM Trace Windows in NS-2
When the trace file is generated, it is ready to be animated by nam. Upon start up, nam will read the trace file, create topology, pop up a window, do layout if necessary and then pause at time 0. Through its user interface, nam provides control over many aspects of animation.

Starting up nam will first create the nam console window. One can have multiple animations running under the same nam insitance. Once a trace file has been loaded into nam, an animation window appears. It has a ‘save layout’ command which saves the current network layout to a file.

7.3.1 Node Configuration Interface

$ns_node-config -addressType hierarchical \ 
  -ad hoc Routing AODV \ 
  -lIType LL \ 
  -macType Mac/802_11 \ 
  -ifqType Queue/DropTail/Priqueue \ 
  -ifqLen 50 \ 
  -antType Antenna/OmiAntenna \ 
  -propType Propagation/TwoRayGround \ 
  -phyType Phy/WirelessPhy \ 
  -topologyInstance Stopo \ 
  -channel Channel/WirelessChannel \ 
-agentTrace ON \n
-routerTrace ON \n
-macTrace OFF \n
-movementTrace OFF\\

7.4 INSTALLING NETWORK SIMULATOR -2

Ns-2.31 network simulator was installed from the website http://www.isi.edu/nsnam/ns/. Installation of ns-2 could be a bit lengthy and a time-consuming process. It involved downloading and setting up a 250 MB package. However, getting the simulator to work was the first step involved in carrying out the simulations.

7.5 PARSING THE SIMULATION TRACE FILES

After each simulation, trace files recording the traffic and node movements are generated. These files need to be parsed in order to extract the information needed to measure the performance metrics. The trace format was used for parsing is as follows:

S 100.000000000 _0_ AGT  --- 20 cbr 40 [0 0 0 0] ------- [0:0 1:0 32 0] [0 0] 0 0

r 100.000000000 _0_ RTR  --- 20 cbr 40 [0 0 0 0] ------- [0:0 1:0 32 0] [0 0] 0 0

Evaluating Packet delivery fraction (PDF)

Packet delivery fraction (PDF %) = (received packets/ sent packets) *100
7.6 SIMULATION PROCEDURE

Run the script by typing at the console

```tcl
ns filename.tcl
```

On completion of the run, the trace output file "filename-out.tr" and nam output file "filename-out.nam" are created. Running filename-out.nam, the mobile nodes moving in nam window can be seen. The number of nodes is set and the percentage of active senders is determined. Now in the animator window, the active senders start informing the network about its presence and begin sending data according to the random progress method.

The finish procedure is given as

```tcl
proc finish {} {
    $ns flush-trace

    close $f

close $nf

    exec nam -r filename.nam &

dexec sleep

    exec ns graph.tcl &

    exit 0
}
```
In the finish procedure, the trace file buffer is cleared and graph are generated in the terminal in a pipelined manner. $nf$ is used to close the trace field. Now the animator field is generated using command

```
exec nam filename.nam
```

and it is used to do the process in the background. Then the procedure is stopped.

To run the file

```
$ns run
```

command is used and the tcl script is executed.