WPI WORCESTER POLYTECHNIC INSTITUTE

Computer Science

NS by Example

Jae Chung

and

Mark Claypool

Purpose

NS (version 2) is an object-oriented, discrete event driven network simulator developed at UC Berkely written in C++ and OTcl. NS is primarily useful for simulating local and wide area networks. Although NS is fairly easy to use once you get to know the simulator, it is quite difficult for a first time user, because there are few user-friendly manuals. Even though there is a lot of documentation written by the developers which has in depth explanation of the simulator, it is written with the depth of a skilled NS user. The purpose of this project is to give a new user some basic idea of how the simultor works, how to setup simulation networks, where to look for further information about network components in simulator codes, how to create new network components, etc., mainly by giving simple examples and brief explanations based on our experiences. Although all the usage of the simulator or possible network simulation setups may not be covered in this project, the project should help a new user to get started quickly.

WPI WORCESTER POLYTECHNIC INSTITUTE

Computer Science

NS by Example

Jae Chung
and
Mark Claypool

Purpose

NS (version 2) is an object-oriented, discrete event driven network simulator developed at UC Berkely written in C++ and OTcl. NS is primarily useful for simulating local and wide area networks. Although NS is fairly easy to use once you get to know the simulator, it is quite difficult for a first time user, because there are few user-friendly manuals. Even though there is a lot of documentation written by the developers which has in depth explanation of the simulator, it is written with the depth of a skilled NS user. The purpose of this project is to give a new user some basic idea of how the simultor works, how to setup simulation networks, where to look for further information about network components in simulator codes, how to create new network components, etc., mainly by giving simple examples and brief explanations based on our experiences. Although all the usage of the simulator or possible network simulation setups may not be covered in this project, the project should help a new user to get started quickly.

Overview

NS is an event driven network simulator developed at UC Berkeley that simulates variety of IP networks. It implements 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 more. NS also implements multicasting and some of the MAC layer protocols for LAN simulations. 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. This document talks briefly about the basic structure of NS, and explains in detail how to use NS mostly by giving examples. Most of the figures that are used in describing the NS basic structure and network components are from the 5th VINT/NS Simulator Tutorial/Workshop slides and the NS Manual (formerly called "NS Notes and Documentation"), modified little bit as needed. For more information about NS and the related tools, visit the VINT project home page.

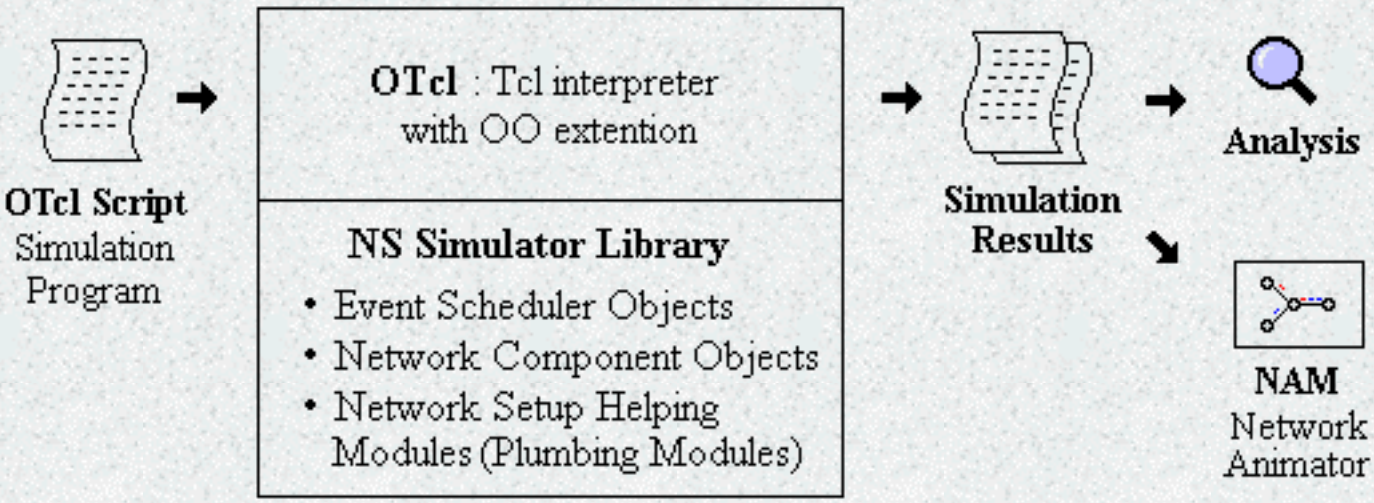

Figure 1. Simplified User's View of NS

As shown in Figure 1, in a simplified user's view, NS is Object-oriented Tcl (OTcl) script interpreter that has a simulation event scheduler and network component object libraries, and network setup (plumbing) module libraries (actually, plumbing modules are implemented as member functions of the base simulator object). In other words, to use NS, you program in OTcl script language. To setup and run a simulation network, a user should write an OTcl script that initiates an event scheduler, 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 "neighbor" 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, which usually are the ones who issued the events, and let them do the appropriate action associated with packet pointed by the event. Network components communicate with one another passing packets, 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 microsecond dequeues 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 that delay does. 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 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 2 shows an object hierarchy example in C++ and OTcl. One thing to note in the figure 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++.

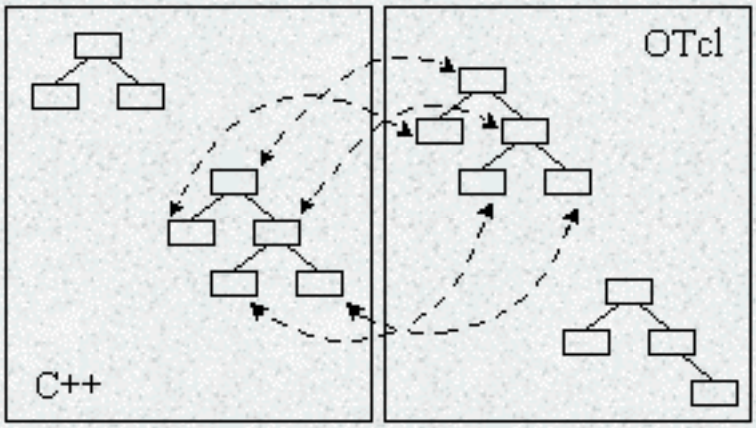

Figure 2. C++ and OTcl: The Duality

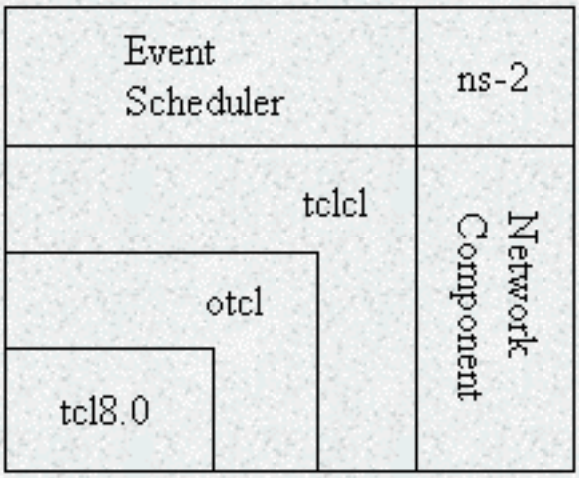

Figure 3. Architectural View of NS

Figure 3 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 tclcl. The whole thing together makes NS, which is a OO extended Tcl interpreter with network simulator libraries.

This section briefly examined the general structure and architecture of NS. At this point, one might be wondering about how to obtain NS simulation results. As shown in Figure 1, 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 (two simulation result analysis 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.

OTcl: The User Language

As mentioned in the overview section, NS is basically an OTcl interpreter with network simulation object libraries. It is very useful to know how to program in OTcl to use NS. This section shows an example Tcl and OTcl script, from which one can get the basic idea of programming in OTcl. These examples are from the 5th VINT/NS Simulation Tutorial/Workshop. This section and the sections after assumes that the reader installed NS, and is familiar with C and C++.

Example 1 is a general Tcl script that shows how to create a procedure and call it, how to assign values to variables, and how to make a loop. Knowing that OTcl is Object-orieneted extension of Tcl, it is obvious that all Tcl commands work on OTcl - the relationship between Tcl and Otcl is just same as C and C++. To run this script you should download ex-tcl.tcl, and type "ns ex-tcl.tcl" at your shell prompt - the command "ns" starts the NS (an OTcl interpreter). You will also get the same results if you type "tcl ex-tcl.tcl", if tcl8.0 is installed in your machine.

```
# Writing a procedure called "test"
proc test {} {
    set a 43
    set b 27
    set c [expr $a + $b]
    set d [expr [expr $a - $b] * $c]
    for {set k 0} {$k < 10} {incr k} {
        if {$k < 5} {
            puts "k < 5, pow = [expr pow($d, $k)]"
        } else {
            puts "k >= 5, mod = [expr $d % $k]"
        }
    }
}

# Calling the "test" procedure created above
test
```

Example 1. A Sample Tcl Script

In Tcl, the keyword **proc** is used to define a procedure, followed by an procedure name and arguments in curly brackets. The keyword **set** is used to assign a value to a variable. [**expr** ...] is to make the interpreter calculate the value of expression within the bracket after the keyword. One thing to note is that to get the value assigned to a variable, **$** is used with the variable name. The keyword **puts** prints out the following string within double quotation marks. The following shows the result of Example 1.

```
k < 5, pow = 1.0
k < 5, pow = 1120.0
k < 5, pow = 1254400.0
k < 5, pow = 1404928000.0
k < 5, pow = 1573519360000.0
k >= 5, mod = 0
k >= 5, mod = 4
k >= 5, mod = 0
k >= 5, mod = 0
k >= 5, mod = 4
```

The next example is an object-oriented programming example in OTcl. This example is very simple, but shows the way which an object is created and used in OTcl. As an ordinary NS user, the chances that you will write your own object might be rare. However, since all of the NS objects that you will use in a NS simulation programming, whether or not they are written in C++ and made available to OTcl via the linkage or written only in OTcl, are essentially OTcl objects, understanding OTcl object is helpful.

```
# add a member function call "greet"
Class mom
mom instproc greet {} {
    $self instvar age_
    puts "$age_ year old mom say:
    How are you doing?"
}

# Create a child class of "mom" called "kid"
# and overide the member function "greet"
Class kid -superclass mom
kid instproc greet {} {
    $self instvar age_
    puts "$age_ year old kid say:
    What's up, dude?"
}

# Create a mom and a kid object, set each age
set a [new mom]
$a set age_ 45
set b [new kid]
$b set age_ 15

# Calling member function "greet" of each object
$a greet
$b greet
```

Example 2. A Sample OTcl Script

Example 2 is an OTcl script that defines two object classes, "mom" and "kid", where "kid" is the child class of "mom", and a member function called "greet" for each class. After the class definitions, each object instance is declared, the "age" variable of each instance is set to 45 (for mom) and 15 (for kid), and the "greet" member function of each object instance is called. The keyword **Class** is to create an object class and **instproc** is to define a member function to an object class. Class inheritance is specified using the keyword **-superclass**. In defining member functions, **$self** acts same as the "this" pointer in C++, and **instvar** checks if the following variable name is already declared in its class or in its superclass. If the variable name given is already declared, the variable is referenced, if not a new one is declared. Finally, to create an object instance, the keyword **new** is used as shown in the example. Downloading ex-otcl.tcl and executing "ns ex-otcl.tcl" will give you the following result:

```
45 year old mom say:
    How are you doing?
15 year old kid say:
    What's up, dude?
```

Simple Simulation Example

This section shows a simple NS simulation script and explains what each line does. Example 3 is an OTcl script that creates the simple network configuration and runs the simulation scenario in Figure 4. To run this simulation, download "ns-simple.tcl" and type "ns ns-simple.tcl" at your shell prompt.

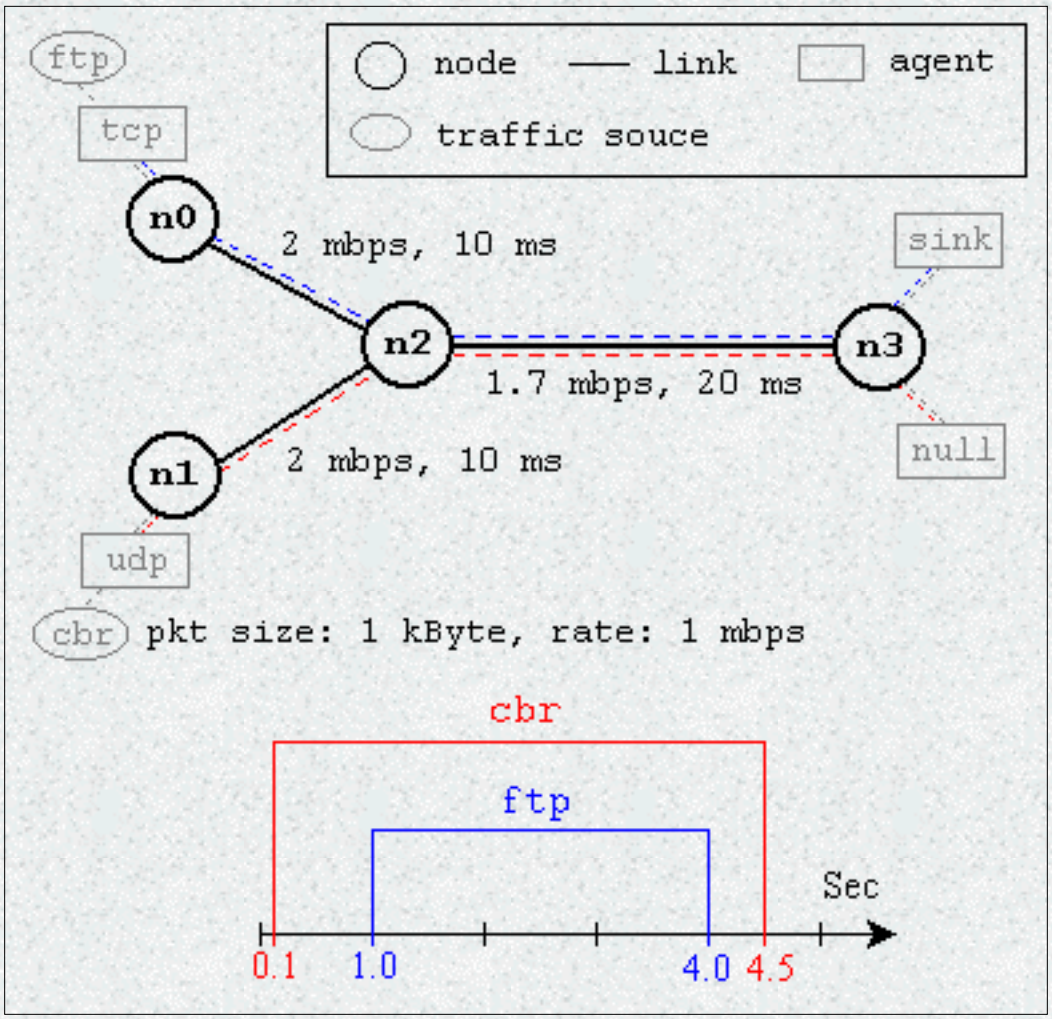

Figure 4. A Simple Network Topology and Simulation Scenario

This network consists of 4 nodes (n0, n1, n2, n3) as shown in above figure. The duplex links between n0 and n2, and n1 and n2 have 2 Mbps of bandwidth and 10 ms of delay. The duplex link between n2 and n3 has 1.7 Mbps of bandwidth and 20 ms of delay. Each node uses a DropTail queue, of which the maximum size is 10. A "tcp" agent is attached to n0, and a connection is established to a tcp "sink" agent attached to n3. As default, the maximum size of a packet that a "tcp" agent can generate is 1KByte. A tcp "sink" agent generates and sends ACK packets to the sender (tcp agent) and frees the received packets. A "udp" agent that is attached to n1 is connected to a "null" agent attached to n3. A "null" agent just frees the packets received. A "ftp" and a "cbr" traffic generator are attached to "tcp" and "udp" agents

respectively, and the "cbr" is configured to generate 1 KByte packets at the rate of 1 Mbps. The "cbr" is set to start at 0.1 sec and stop at 4.5 sec, and "ftp" is set to start at 1.0 sec and stop at 4.0 sec.

```
#Create a simulator object
set ns [new Simulator]

#Define different colors for data flows (for NAM)
$ns color 1 Blue
$ns color 2 Red

#Open the NAM trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {} {
        global ns nf
        $ns flush-trace
        #Close the NAM trace file
        close $nf
        #Execute NAM on the trace file
        exec nam out.nam &
        exit 0
}

#Create four nodes
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

#Create links between the nodes
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
$ns duplex-link $n2 $n3 1.7Mb 20ms DropTail

#Set Queue Size of link (n2-n3) to 10
$ns queue-limit $n2 $n3 10

#Give node position (for NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

#Monitor the queue for link (n2-n3). (for NAM)
$ns duplex-link-op $n2 $n3 queuePos 0.5
```

```
#Setup a TCP connection
set tcp [new Agent/TCP]
$tcp set class_ 2
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink
$tcp set fid_ 1

#Setup a FTP over TCP connection
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

#Setup a UDP connection
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $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 type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb
$cbr set random_ false


#Schedule events for the CBR and FTP agents
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"

#Detach tcp and sink agents (not really necessary)
$ns at 4.5 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"

#Call the finish procedure after 5 seconds of simulation time
$ns at 5.0 "finish"

#Print CBR packet size and interval
puts "CBR packet size = [$cbr set packet_size_]"
puts "CBR interval = [$cbr set interval_]"

#Run the simulation
```

```
$ns run
```

Example 3. A Simple NS Simulation Script

The following is the explanation of the script above. In general, an NS script starts with making a Simulator object instance.

- set *ns* [new Simulator]: generates an NS simulator object instance, and assigns it to variable *ns* (italics is used for variables and values in this section). What this line does is the following:
  - Initialize the packet format (ignore this for now)
  - Create a scheduler (default is calendar scheduler)
  - Select the default address format (ignore this for now)

  The "Simulator" object has member functions that do the following:
  - Create compound objects such as nodes and links (described later)
  - Connect network component objects created (ex. attach-agent)
  - Set network component parameters (mostly for compound objects)
  - Create connections between agents (ex. make connection between a "tcp" and "sink")
  - Specify NAM display options
  - Etc.

  Most of member functions are for simulation setup (referred to as plumbing functions in the Overview section) and scheduling, however some of them are for the NAM display. The "Simulator" object member function implementations are located in the "ns-2/tcl/lib/ns-lib.tcl" file.

- *$ns* color *fid color*: is to set color of the packets for a flow specified by the flow id (fid). This member function of "Simulator" object is for the NAM display, and has no effect on the actual simulation.

- *$ns* namtrace-all *file-descriptor*: This member function tells the simulator to record simulation traces in NAM input format. It also gives the file name that the trace will be written to later by the command *$ns* flush-trace. Similarly, the member function trace-all is for recording the simulation trace in a general format.

- proc *finish* {}: is called after this simulation is over by the command *$ns* at 5.0 "*finish*". In this function, post-simulation processes are specified.

- set *n0* [*$ns* node]: The member function node creates a node. A node in NS is compound object made of address and port classifiers (described in a later section). Users can create a node by separately creating an address and a port classifier objects and connecting them together. However, this member function of Simulator object makes the job easier. To see how a node is created, look

at the files: "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-node.tcl".

- *$ns* duplex-link *node1 node2 bandwidth delay queue-type*: creates two simplex links of specified bandwidth and delay, and connects the two specified nodes. In NS, the output queue of a node is implemented as a part of a link, therefore users should specify the queue-type when creating links. In the above simulation script, DropTail queue is used. If the reader wants to use a RED queue, simply replace the word DropTail with RED. The NS implementation of a link is shown in a later section. Like a node, a link is a compound object, and users can create its sub-objects and connect them and the nodes. Link source codes can be found in "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-link.tcl" files. One thing to note is that you can insert error modules in a link component to simulate a lossy link (actually users can make and insert any network objects). Refer to the NS documentation to find out how to do this.

- *$ns* queue-limit *node1 node2 number*: This line sets the queue limit of the two simplex links that connect node1 and node2 to the number specified. At this point, the authors do not know how many of these kinds of member functions of Simulator objects are available and what they are. Please take a look at "ns-2/tcl/libs/ns-lib.tcl" and "ns-2/tcl/libs/ns-link.tcl", or NS documentation for more information.

- *$ns* duplex-link-op *node1 node2 ...*: The next couple of lines are used for the NAM display. To see the effects of these lines, users can comment these lines out and try the simulation.

Now that the basic network setup is done, the next thing to do is to setup traffic agents such as TCP and UDP, traffic sources such as FTP and CBR, and attach them to nodes and agents respectively.

- set *tcp* [new *Agent/TCP*]: This line shows how to create a TCP agent. But in general, users can create any agent or traffic sources in this way. Agents and traffic sources are in fact basic objects (not compound objects), mostly implemented in C++ and linked to OTcl. Therefore, there are no specific Simulator object member functions that create these object instances. To create agents or traffic sources, a user should know the class names these objects (Agent/TCP, Agnet/TCPSink, Application/FTP and so on). This information can be found in the NS documentation or partly in this documentation. But one shortcut is to look at the "ns-2/tcl/libs/ns-default.tcl" file. This file contains the default configurable parameter value settings for available network objects. Therefore, it works as a good indicator of what kind of network objects are available in NS and what are the configurable parameters.

- *$ns* attach-agent *node agent*: The attach-agent member function attaches an agent object created to a node object. Actually, what this function does is call the attach member function of specified node, which attaches the given agent to itself. Therefore, a user can do the same thing by, for example, $n0 attach $tcp. Similarly, each agent object has a member function attach-agent that attaches a traffic source object to itself.

- *$ns* connect *agent1 agent2*: After two agents that will communicate with each other are created, the next thing is to establish a logical network connection between them. This line establishes a network connection by setting the destination address to each others' network and port address

pair.

Assuming that all the network configuration is done, the next thing to do is write a simulation scenario (i.e. simulation scheduling). The Simulator object has many scheduling member functions. However, the one that is mostly used is the following:

- *$ns* at *time "string"*: This member function of a Simulator object makes the scheduler (scheduler_ is the variable that points the scheduler object created by [new Scheduler] command at the beginning of the script) to schedule the execution of the specified string at given simulation time. For example, *$ns* at *0.1 "$cbr start"* will make the scheduler call a start member function of the CBR traffic source object, which starts the CBR to transmit data. In NS, usually a traffic source does not transmit actual data, but it notifies the underlying agent that it has some amount of data to transmit, and the agent, just knowing how much of the data to transfer, creates packets and sends them.

After all network configuration, scheduling and post-simulation procedure specifications are done, the only thing left is to run the simulation. This is done by *$ns* run.

Event Scheduler

This section talks about the discrete event schedulers of NS. 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 5 shows each network object using an event scheduler. Note that a network object that issues an event is the one who handles the event later at scheduled time. Also note that the data path between network objects is different from the event path. Actually, packets are handed from one network object to another using *send(Packet* p) {target_->recv(p)};* method of the sender and *recv(Packet*, Handler* h = 0)* method of the receiver.

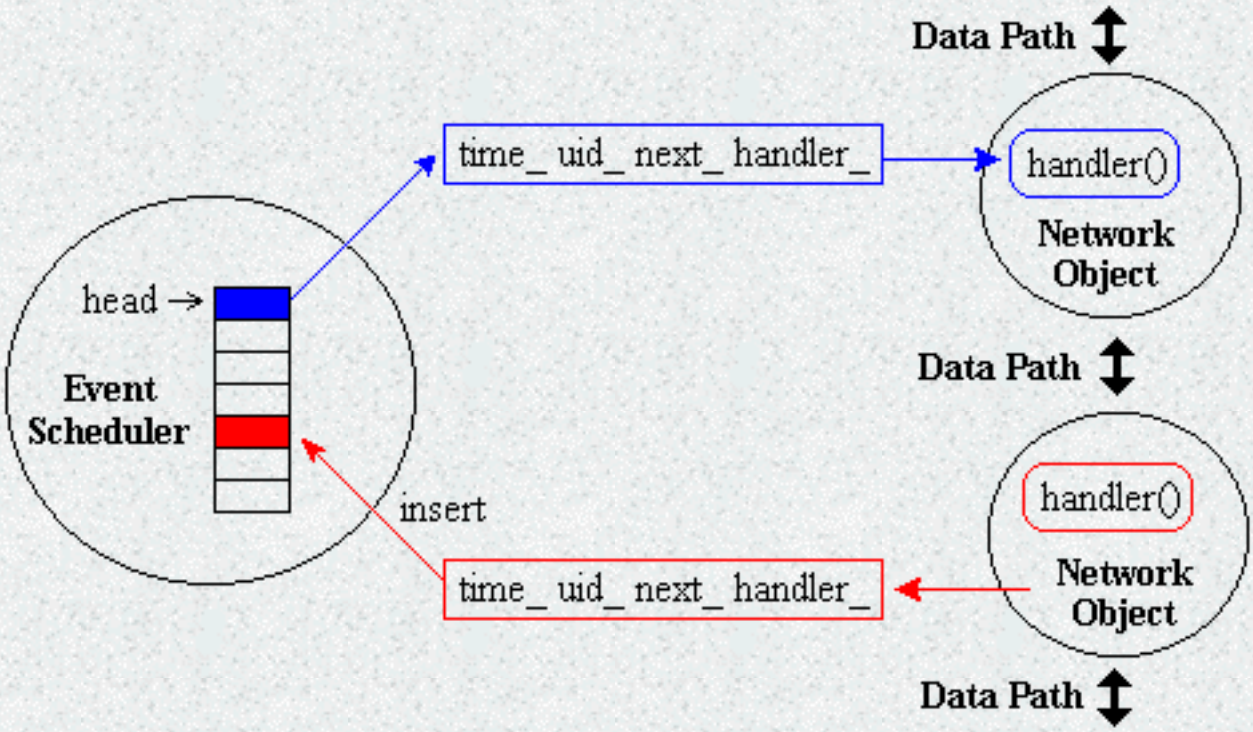

Figure 5. Discrete Event Scheduler

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 are 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 hacking around the internals. The Calendar non-real-time scheduler is set as the default. The real-time scheduler is for emulation, which allow the simulator to interact with a real network. Currently, emulation is under development although an experimental version is available. The following is an example of selecting a specific event scheduler:

```
. . .
set ns [new Simulator]
$ns use-scheduler Heap
. . .
```

Another use of an event scheduler is to schedule simulation events, such as when to start an FTP application, when to finish a simulation, or for simulation scenario generation prior to a simulation run. An event scheduler object itself has simulation scheduling member functions such as at *time "string"* that issue a special event called AtEvent at a specified simulation *time*. An "AtEvent" is actually a child class of "Event", which has an additional variable to hold the given *string*. However, it is treated the same as a normal (packet related) event within the event scheduler. When a simulation is started, and as the scheduled time for an AtEvent in the event queue comes, the AtEvent is passed to an "AtEvent handler" that is created once and handles all AtEvents, and the OTcl command specified by the *string* field of the AtEvent is executed. The following is a simulation event scheduling line added version of the above example.

```
. . .
set ns [new Simulator]
$ns use-scheduler Heap
$ns at 300.5 "complete_sim"
. . .

proc complete_sim {} {
. . .
}
```

You might noticed from the above example that at *time "string"* is a member function of the Simulator object (set *ns* [new Simulator]). But remember that the Simulator object just acts as a user interface, and it actually calls the member functions of network objects or a scheduler object that does the real job. Followings are a partial list and brief description of Simulator object member functions that interface with scheduler member functions:

```
Simulator instproc now          # return scheduler's notion of current time
Simulator instproc at args      # schedule execution of code at specified time
Simulator instproc at-now args  # schedule execution of code at now
Simulator instproc after n args # schedule execution of code after n secs
Simulator instproc run args     # start scheduler
Simulator instproc halt         # stop (pause) scheduler
```

Network Components

This section talks about the NS components, mostly compound network components. Figure 6 shows a partial OTcl class hierarchy of NS, which will help understanding the basic network components. For a complete NS class hierarchy, visit http://www-sop.inria.fr/rodeo/personnel/Antoine.Clerget/ns.

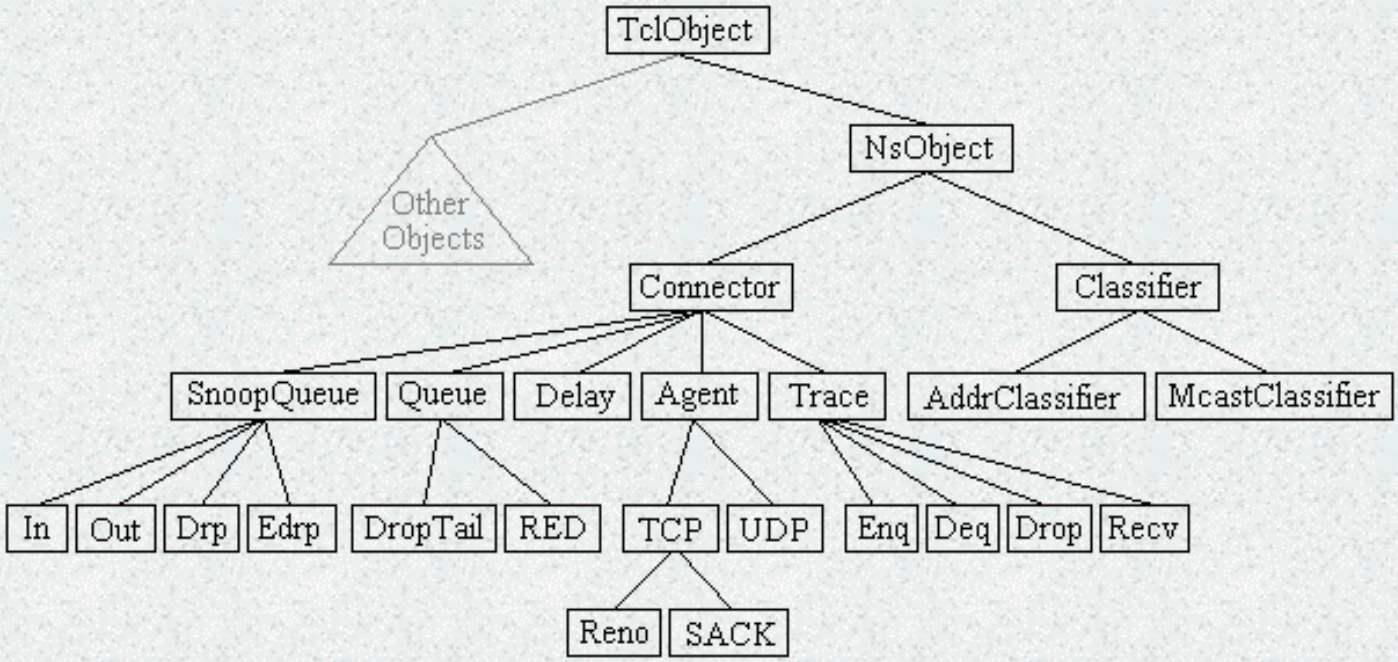

Figure 6. Class Hierarchy (Partial)

The root of the hierarchy is the TclObject class that is the superclass of all OTcl library objects (scheduler, network components, timers and the other objects including NAM related ones). As an ancestor class of TclObject, NsObject class is the superclass of all basic network component objects that handle packets, which may compose compound network objects such as nodes and links. The basic network components are further divided into two subclasses, Connector and Classifier, based on the number of the possible output data paths. The basic network objects that have only one output data path are under the Connector class, and switching objects that have possible multiple output data paths are under the Classifier class.

- **Node and Routing**

A node is a compound object composed of a node entry object and classifiers as shown in Figure 7. There are two types of nodes in NS. A unicast node has an address classifier that does unicast routing and a port classifier. A multicast node, in addition, has a classifier that classify multicast packets from unicast packets and a multicast classifier that performs multicast routing.

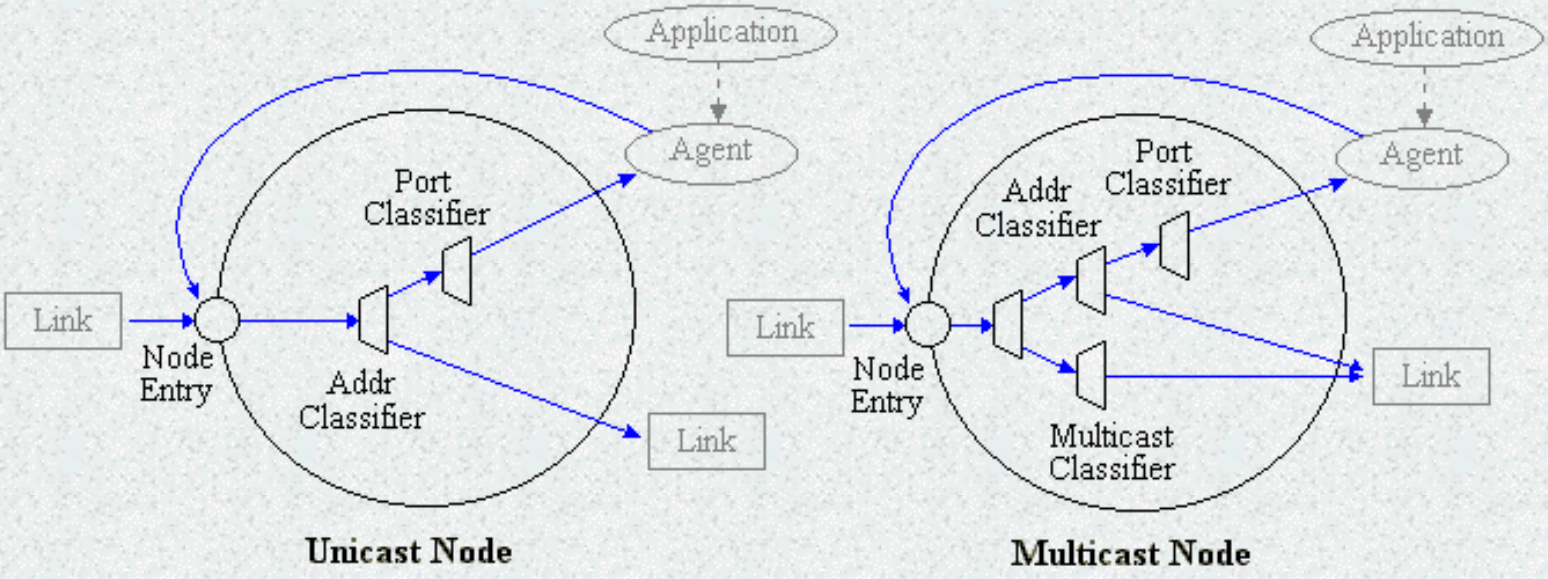

Figure 7. Node (Unicast and Multicast)

In NS, Unicast nodes are the default nodes. To create Multicast nodes the user must explicitly notify in the input OTcl script, right after creating a scheduler object, that all the nodes that will be created are multicast nodes. After specifying the node type, the user can also select a specific routing protocol other than using a default one.

- **Unicast**
  - *$ns* rtproto *type*
  - *type*: Static, Session, DB, cost, multi-path

- **Multicast**
  - *$ns* multicast (right after set *$ns* [new Scheduler])
  - *$ns* mrtproto *type*
  - *type*: CtrMcast, DM, ST, BST

For more information about routing, refer to the NS Manual located at http://www.isi.edu/nsnam/ns/ns-documentation.html. The documentation has chapters talk about unicast and multicast routing.

- **Link**

A link is another major compound object in NS. When a user creates a link using a duplex-link member function of a Simulator object, two simplex links in both directions are created as shown in Figure 8.

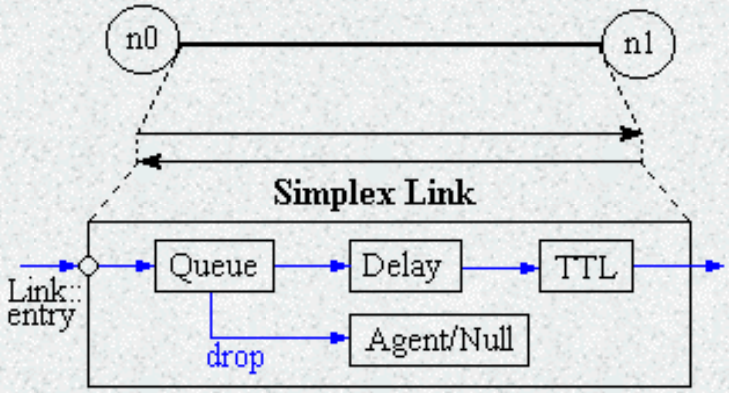

Figure 8. Link

One thing to note is that an output queue of a node is actually implemented as a part of simplex link object. Packets dequeued from a queue are passed to the Delay object that simulates the link delay, and packets dropped at a queue are sent to a Null Agent and are freed there. Finally, the TTL object calculates Time To Live parameters for each packet received and updates the TTL field of the packet.

- **Tracing**

  In NS, network activities are traced around simplex links. If the simulator is directed to trace network activities (specified using *$ns* trace-all *file* or *$ns* namtrace-all *file*), the links created after the command will have the following trace objects inserted as shown in Figure 9. Users can also specifically create a trace object of type *type* between the given *src* and *dst* nodes using the create-trace {*type file src dst*} command.

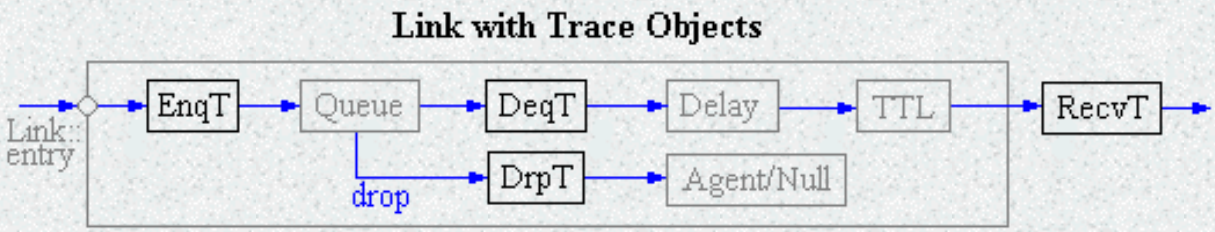

Figure 9. Inserting Trace Objects

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.

- **Queue Monitor**

  Basically, tracing objects are designed to record packet arrival time at which they are located. Although a user gets enough information from the trace, he or she might be interested in what is going on inside a specific output queue. For example, a user interested in RED queue behavior may want to measure the dynamics of average queue size and current queue size of a specific RED queue (i.e. need for queue monitoring). Queue monitoring can be achieved using queue monitor objects and snoop queue objects as shown in Figure 10.

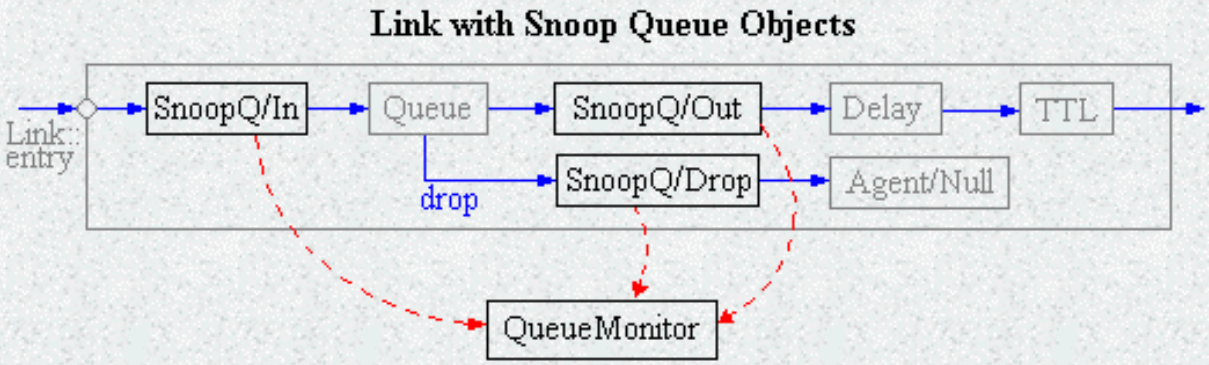

Figure 10. Monitoring Queue

When a packet arrives, a snoop queue object notifies the queue monitor object of this event. The queue monitor using this information monitors the queue. A RED queue monitoring example is shown in the RED Queue Monitor Example section. Note that snoop queue objects can be used in parallel with tracing objects even though it is not shown in the above figure.

- **Packet Flow Example**

Until now, the two most important network components (node and link) were examined. Figure 11 shows internals of an example simulation network setup and packet flow. The network consist of two nodes (n0 and n1) of which the network addresses are 0 and 1 respectively. A TCP agent attached to n0 using port 0 communicates with a TCP sink object

attached to n1 port 0. Finally, an FTP application (or traffic source) is attached to the TCP agent, asking to send some amount of data.

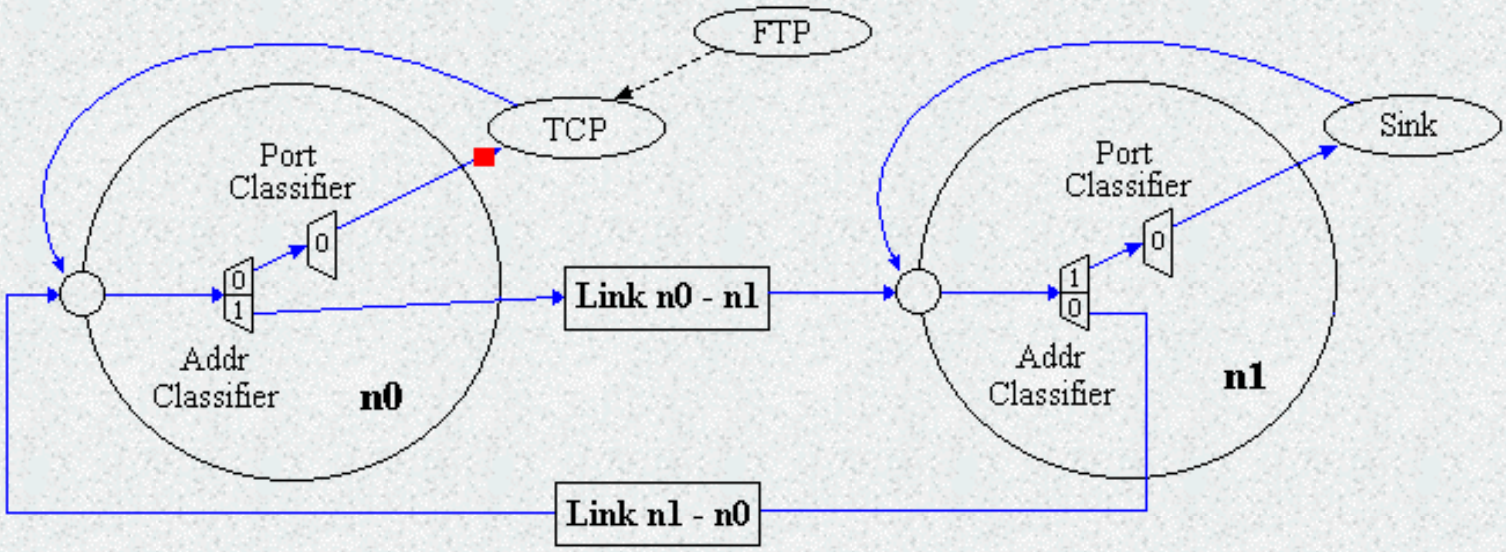

Figure 11. Packet Flow Example

Note that the above figure does not show the exact behavior of a FTP over TCP. It only shows the detailed internals of simulation network setup and a packet flow.

Packet

A NS packet is composed of a stack of headers, and an optional data space (see Figure 12). 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. What this means is 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, and a network object can access any header in the stack of a packet it processes using the corresponding offset value.

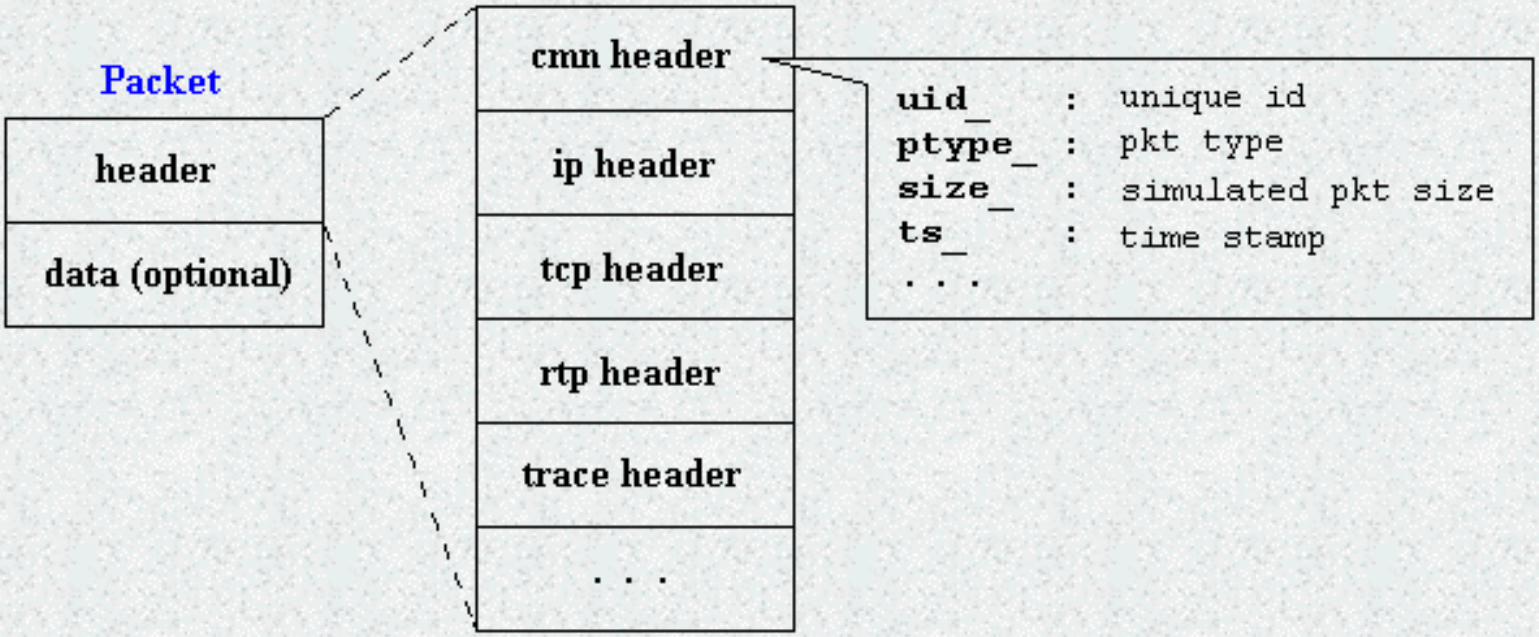

Figure 12. NS Packet Format

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. However, if you want to implement an application that talks to another application cross the network, you might want to use this feature with a little modification in the underlying agent implementation. Another possible approach would be creating a new header for the application and modifying the underlying agent to write data received from the application to the new header. The second approach is shown as an example in a later section called "Add New Application and Agent".

Trace Analysis Example

This section shows a trace analysis example. Example 4 is the same OTcl script as the one in the "Simple Simulation Example" section with a few lines added to open a trace file and write traces to it. For the network topology it generates and the simulation scenario, refer to Figure 4 in the "Simple Simulation Example" section. To run this script download "ns-simple-trace.tcl" and type "ns ns-simple-trace.tcl" at your shell prompt.

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

#Open the Trace file
set tf [open out.tr w]
$ns trace-all $tf

#Define a 'finish' procedure
proc finish {} {
        global ns nf tf
        $ns flush-trace
        #Close the NAM trace file
        close $nf
        #Close the Trace file
        close $tf
        #Execute NAM on the trace file
        exec nam out.nam &
        exit 0
}

. . .
```

Example 4. Trace Enabled Simple NS Simulation Script (modified from Example 3)

Running the above script generates a NAM trace file that is going to be used as an input to NAM and a trace file called "out.tr" that will be used for our simulation analysis. Figure 13 shows the trace format and example trace data from "out.tr".

event	time	from node	to node	pkt type	pkt size	flags	fid	src addr	dst addr	seq num	pkt id

```
r : receive (at to_node)
+ : enqueue (at queue)                src_addr : node.port (3.0)
- : dequeue (at queue)                dst_addr : node.port (0.0)
d : drop    (at queue)
```

```
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 13. Trace Format Example

Each trace line starts with an event (+, -, d, r) descriptor followed by the simulation time (in seconds) of that event, and from and to node, which identify the link on which the event occurred. Look at Figure 9 in the "Network Components" section to see where in a link each type of event is traced. The next information in the line before flags (appeared as "------" since no flag is set) is packet type and size (in Bytes). Currently, NS implements only the Explicit Congestion Notification (ECN) bit, and the remaining bits are not used. The next field is flow id (fid) of IPv6 that a user can set for each flow at the input OTcl script. Even though fid field may not used in a simulation, users can use this field for analysis purposes. The fid field is also used when specifying stream color for the NAM display. The next two fields are source and destination address in forms of "node.port". The next field shows the network layer protocol's packet sequence number. Note that even though UDP implementations do not use sequence number, NS keeps track of UDP packet sequence number for analysis purposes. The last field shows the unique id of the packet.

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. To make the example work on your machine, you should download "column" and make it executable (i.e. "chmod 755 column"). 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".

```
cat out.tr | grep " 2 3 cbr " | grep ^r | column 1 10 | awk '{dif = $2 - old2;
if(dif==0) dif = 1; if(dif > 0) {printf("%d\t%f\n", $2, ($1 - old1) / dif);
old1 = $1; old2 = $2}}' > jitter.txt
```

This shell command selects the "CBR packet receive" event at n3, selects time (column 1) and sequence number (column 10), and calculates the difference from last packet receive time divided by difference in sequence number (for loss packets) for each sequence number. The following is the corresponding jitter

graph that is generated using gnuplot. The X axis show the packet sequence number and the Y axis shows simulation time in seconds.

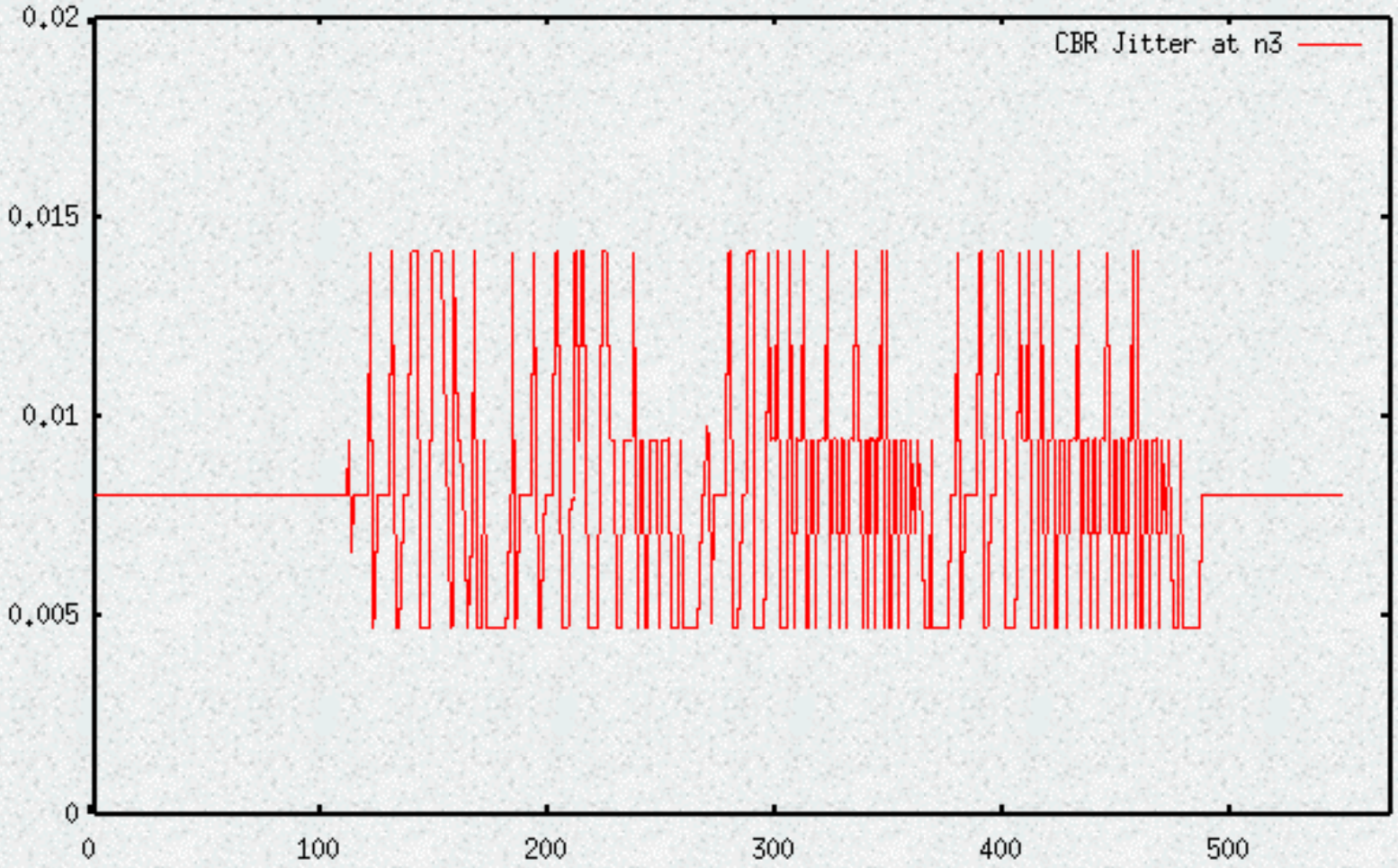

Figure 14. CBR Jitter at The Receiving Node (n3)

You might also check for more utilities in the Example Utilities section.

This section showed an example of how to generate traces in NS, how to interpret them, and how to get useful information out from the traces. In this example, the post simulation processes are done in a shell prompt after the simulation. However, these processes can be included in the input OTcl script, which is shown in the next section.

RED Queue Monitor Example

This section shows a RED queue monitoring example. Example 5 is an OTcl script written by Polly Huang, which sets up the network topology and runs the simulation scenario shown in Figure 15. Note that a RED queue that can hold up to 25 packets is used for the link r1-r2, and we want to see how the RED queue works by measuring the dynamics of current and average queue size. To run this script, download "red.tcl" and type "ns red.tcl" at your shell prompt.

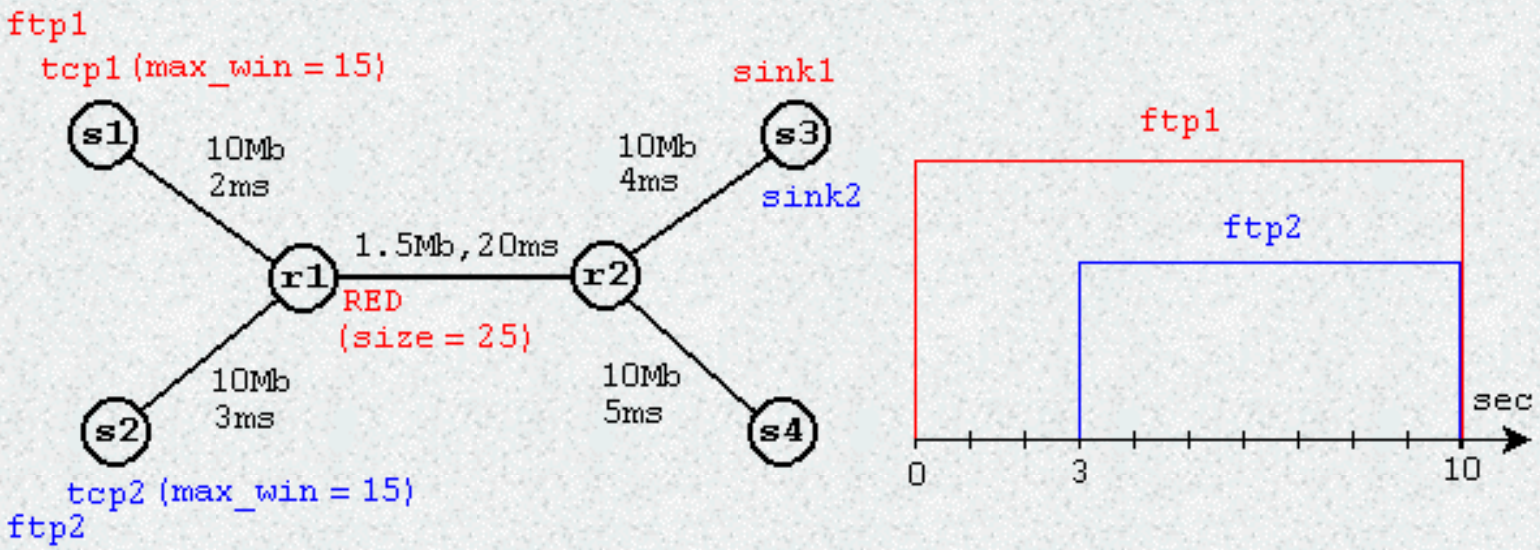

Figure 15. RED Queue Monitor Example Setup

```
set ns [new Simulator]

set node_(s1) [$ns node]
set node_(s2) [$ns node]
set node_(r1) [$ns node]
set node_(r2) [$ns node]
set node_(s3) [$ns node]
set node_(s4) [$ns node]

$ns duplex-link $node_(s1) $node_(r1) 10Mb 2ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 10Mb 3ms DropTail
$ns duplex-link $node_(r1) $node_(r2) 1.5Mb 20ms RED
$ns queue-limit $node_(r1) $node_(r2) 25
$ns queue-limit $node_(r2) $node_(r1) 25
$ns duplex-link $node_(s3) $node_(r2) 10Mb 4ms DropTail
$ns duplex-link $node_(s4) $node_(r2) 10Mb 5ms DropTail
  .
  .
  .
```

```
set tcp1 [$ns create-connection TCP/Reno $node_(s1) TCPSink $node_(s3) 0]
$tcp1 set window_ 15
set tcp2 [$ns create-connection TCP/Reno $node_(s2) TCPSink $node_(s3) 1]
$tcp2 set window_ 15
set ftp1 [$tcp1 attach-source FTP]
set ftp2 [$tcp2 attach-source FTP]

# Tracing a queue
set redq [[$ns link $node_(r1) $node_(r2)] queue]
set tchan_ [open all.q w]
$redq trace curq_
$redq trace ave_
$redq attach $tchan_


$ns at 0.0 "$ftp1 start"
$ns at 3.0 "$ftp2 start"
$ns at 10 "finish"

# Define 'finish' procedure (include post-simulation processes)
proc finish {} {
    global tchan_
    set awkCode {
        {
            if ($1 == "Q" && NF>2) {
                print $2, $3 >> "temp.q";
                set end $2
            }
            else if ($1 == "a" && NF>2)
            print $2, $3 >> "temp.a";
        }
    }
    set f [open temp.queue w]
    puts $f "TitleText: red"
    puts $f "Device: Postscript"

    if { [info exists tchan_] } {
        close $tchan_
    }
    exec rm -f temp.q temp.a
    exec touch temp.a temp.q

    exec awk $awkCode all.q

    puts $f \"queue
    exec cat temp.q >@ $f
    puts $f \n\"ave_queue
    exec cat temp.a >@ $f
    close $f
```

```
    exec xgraph -bb -tk -x time -y queue temp.queue &
    exit 0
}

$ns run
```

Example 5. RED Queue Monitor Simulation Script

There are couple of things to note in the above script. First, more advanced Simulator object member function create-connection is used to create TCP connections. Second, take a good look at queue tracing (monitoring) part of the script. These lines make a variable to point the RED queue object, call its member function trace to monitor the current queue size (curq_) and average queue size (avg_), and make them write the results to the file "all.q". Following are the two queue trace output formats for average queue size and current queue size.

```
a time avg_q_size
Q time crnt_q_size
```

Now that the all commands needed for RED queue monitoring is done except closing the "all.q" file when the simulation is done, the remaining work designs a post-simulation process. This script, using awk, does this process of transforming the monitored information into XGraph (a plotter) input format, and launches it to draw the given information (see Figure 16).

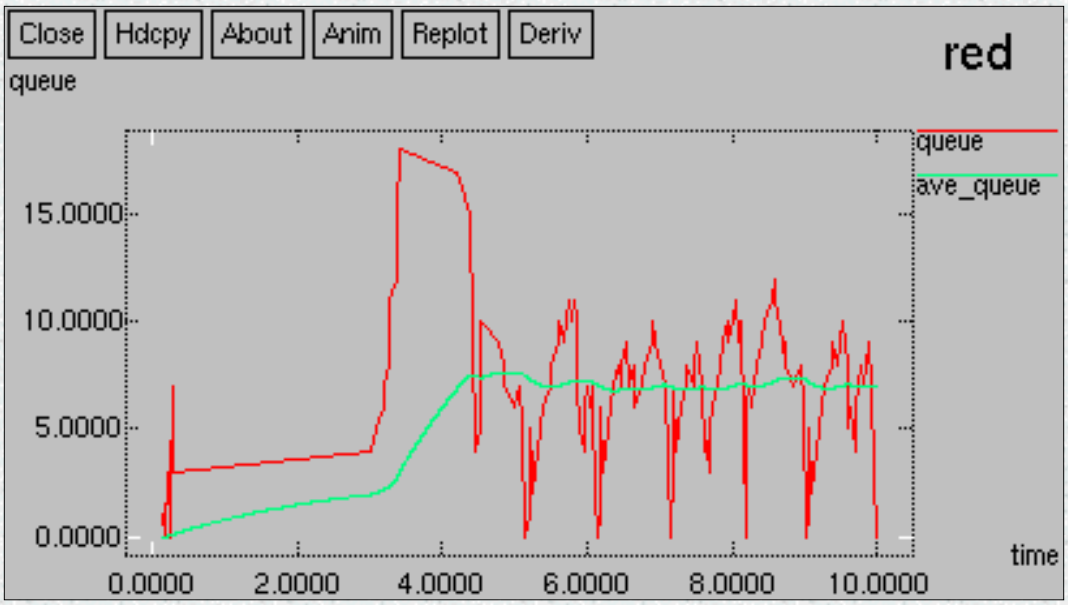

Figure 16. Red Queue Trace Graph

Trace Analysis Utilities

Here is a collection of trace analysis utilities that have been useful:

- Much parsing is easy if you can extract columns. Here is a helper-perl utility called "column" that breaks a text output into columns, separated by whitespace. It is used by many of the scripts below.
- Here is a simple perl called stats.pl script that prints some stats along the bottleneck link (node 1 to node 2, in this case) of an output file stats.tr. Run it with the command:
  `stats.pl -l1 1 -l2 2 -max 1.5 stats.tr`.
- Here is a tunneled shell command called jitter.sh 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". This shell command selects the "CBR packet receive" event at n3, selects time (column 1) and sequence number (column 10), and calculates the difference from last packet receive time divided by difference in sequence number (for loss packets) for each sequence number.

Where to Find What?

Before going into a discussion of how to extend NS, let's briefly examine what information is stored in which directory or file. Figure 17 shows a part of the directory structure of the simulator if you installed it using the ns-allinone-2.1b package.

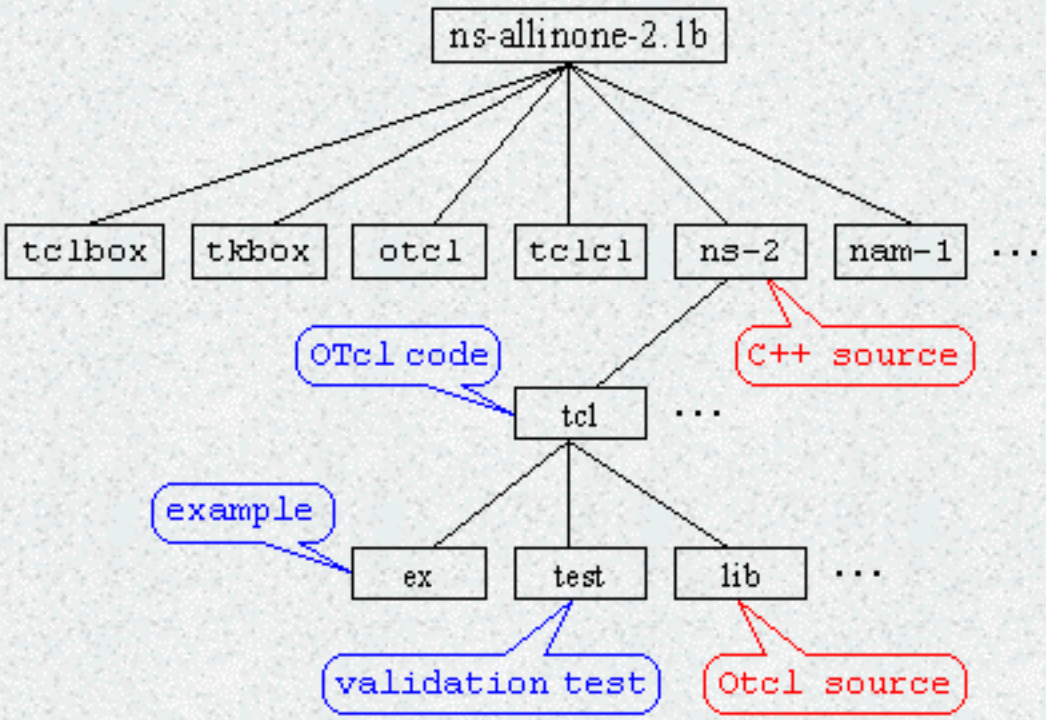

Figure 17. NS Directory Structure

Among the sub-directories of ns-allinone-2.1b, ns-2 is the place that has all of the simulator implementations (either in C++ or in OTcl), validation test OTcl scripts and example OTcl scripts. Within this directory, all OTcl codes and test/example scripts are located under a sub-directory called tcl, and most of C++ code, which implements event scheduler and basic network component object classes, except the WWW related ones, are located in the main level. For example, if you want to see the implementation of the UDP agent, you should go to "ns-allinone-2.1b/ns-2" directory, and open up "udp.h", "udp.cc" and the files that contain the implementation of ancestor classes of UDP as needed. For the class hierarchy of network components, refer to Figure 6 in the "Network Components" section. From now on, it is assumed that you are already in "ns-allinone-2.1b" directory.

The tcl directory has sub-directories, among which the lib directory that contains OTcl source codes for the most basic and essential parts of the NS implementation (agent, node, link, packet, address, routing, and etc.) is the place one as a user or as a developer will visit the most. Note that the OTcl source codes for LAN, Web, and Multicast implementations are located in separate subdirectories of tcl. Following is

a partial list of files in "ns-2/tcl/lib" directory.

- ns-lib.tcl: The simulator class and most of its member function definitions except for LAN, Web, and Multicast related ones are located here. If you want to know which member functions of the Simulator object class are available and how they work, this is a place to look.

- ns-default.tcl: The default values for configurable parameters for various network components are located here. Since most of network components are implemented in C++, the configurable parameters are actually C++ variables made available to OTcl via an OTcl linkage function, bind(*C++_variable_name*, *OTcl_variable_name*). How to make OTcl linkages from C++ code is described in the next section.

- ns-packet.tcl: The packet header format initialization implementation is located here. When you create a new packet header, you should register the header in this file to make the packet header initialization process to include your header into the header stack format and give you the offset of your header in the stack. A new header creating example is shown in "Add New Application and Agent" section.

- other OTcl files: Other OTcl files in this directory, contain OTcl implementation of compound network objects or the front end (control part) of network objects in C++. The FTP application is entirely implemented in OTcl and the source code is located in "ns-source.tcl".

Two sub-directories of tcl that might be interesting for a user who wants to know how to design a specific simulation are ex and test. The former directory contains various example simulation scripts and the latter contains simulation scripts that validate the NS installed in your machine by running various simulations and comparing the results with the expected results.

OTcl Linkage

Extending NS by adding a new basic network object usually involves working around making OTcl linkage from C++ code, since the object class in the data path should be written in C++ for efficiency reason. This section introduces C++/OTcl linkages available in NS by giving an example of creating a simple and dull agent called "MyAgent" that has no behavior of an agent (i.e. no packet creation and transmission). Figures 18 to 21 show parts of the C++ source file for "MyAgent", together which makes a complete implementation (with 3 extra header lines). Also, an OTcl script with which you can test "MyAgent" is presented at the end of this section.

- **Export C++ class to OTcl**

Suppose that you created a new network object class in C++, say "MyAgent" that is derived from the "Agent" class, and want to make it possible to create an instance of this object in OTcl. To do this you have to define a linkage object, say "MyAgentClass", which should be derived from "TclClass". This linkage object creates an OTcl object of specified name ("Agent/MyAgentOtcl" in this example), and creates a linkage between the OTcl object and the C++ object ("MyAgent" in this example), of which the instance launching procedure is specified in the "**create**" member function. Figure 18 shows the "MyAgent" class definition and the linkage class definition.

```
class MyAgent : public Agent {
public:
        MyAgent();
protected:
        int command(int argc, const char*const* argv);
private:
        int    my_var1;
        double my_var2;
        void   MyPrivFunc(void);
};
```

```
static class MyAgentClass : public TclClass {
public:
        MyAgentClass() : TclClass("Agent/MyAgentOtcl") {}
        TclObject* create(int, const char*const*) {
                return(new MyAgent());
        }
} class_my_agent;
```

Figure 18. Example C++ Network Component and The Linkage Object

When NS is first started, it executes the constructor for the static variable "*class_my_agent*", and thus an instance of "MyAgentClass" is created. In this process, the "Agent/MyAgentOtcl" class and its appropriate methods (member functions) are created in OTcl space. Whenever a user in OTcl space tries to create an instance of this object using the command `"new Agent/MyAgentOtcl"`, it invokes "MyAgentClass::create" that creates an instance of "MyAgent" and returns the address. Note that creating a C++ object instance from OTcl does not mean that you can invoke member functions or access member variables of the C++ object instance from OTcl.

- **Export C++ class variables to OTcl**

Suppose your new C++ object, "MyAgent", has two parameter variables, say "**my_var1**" and "**my_var2**", that you want to configure (or change) easily from OTcl using the input simulation script. To do this you should use a binding function for each of the C++ class variables you want to export. A binding function creates a new member variable of given name (first argument) in the matching OTcl object class ("Agent/MyAgentOtcl"), and create bi-directonal bindings between the OTcl class variable and the C++ variable whose address is specified as the second variable. Figure 19 shows how to make bindings for "my_var1" and "my_var2" shown in Figure 18.

```
MyAgent::MyAgent() : Agent(PT_UDP) {
        bind("my_var1_otcl", &my_var1);
        bind("my_var2_otcl", &my_var2);
}
```

Figure 19. Variable Binding Creation Example

Note that the binding functions are placed in the "MyAgent" constructor function to establish the bindings when an instance of this object is created. NS support four different binding functions for five different variable types as follows:

```
- bind(): real or integer variables
- bind_time(): time variable
- bind_bw(): bandwidth variable
- bind_bool(): boolean variable
```

In this way, a user designing and running a simulation using an OTcl script can change or access configurable parameters (or variable values) of network components implemented in C++. Note that whenever you export a C++ variable, it is recommended that you also set the default value for that variable in the "ns-2/tcl/lib/ns-lib.tcl" file. Otherwise, you will get a warning message when you create an instant of your new object.

- **Export C++ Object Control Commands to OTcl.**

In addition to exporting some of your C++ object variables, you may also want to give the control of your C++ object to OTcl. This is done by defining a "**command**" member function of your C++ object ("MyAgent"), which works as an OTcl command interpreter. As a matter of fact, an OTcl command

defined in a "command" member function of a C++ object looks the same as a member function of the matching OTcl object to a user. Figure 20 shows an example "command" member function definition for the "MyAgent" object in Figure 18.

```
int MyAgent::command(int argc, const char*const* argv) {
        if(argc == 2) {
                if(strcmp(argv[1], "call-my-priv-func") == 0) {
                        MyPrivFunc();
                        return(TCL_OK);
                }
        }
        return(Agent::command(argc, argv));
}
```

Figure 20. Example OTcl command interpreter

When an instance of the shadow OTcl that matches the "MyAgent" object is created in OTcl space (i.e. `set myagent [new Agent/MyAgentOtcl]`), and the user tries to call a member function of that object (i.e. `$myagent call-my-priv-func`), OTcl searches for the given member function name in the OTcl object. If the given member function name cannot be found, then it invokes the "MyAgent::**command**" passing the invoked OTcl member function name and arguments in argc/argv format. If there is an action defined for the invoked OTcl member function name in the "command" member function, it carries out what is asked and returns the result. If not, the "command" function for its ancestor object is recursively called until the name is found. If the name cannot be found in any of the ancestors, an error message is return to the OTcl object, and the OTcl object gives an error message to the user. In this way, an user in OTcl space can control a C++ object's behavior.

- **Execute an OTcl command from C++.**

As you implement a new network object in C++, you might want to execute an OTcl command from the C++ object. Figure 21 shows the implementation of "**MyPrivFunc**" member function of "MyAgent" in Figure 18, which makes an OTcl interpreter print out the value in "**my_var1**" and "**my_var2**" private member variables.

```
void MyAgent::MyPrivFunc(void) {
        Tcl& tcl = Tcl::instance();
        tcl.eval("puts \"Message From MyPrivFunc\"");
        tcl.evalf("puts \"      my_var1 = %d\"", my_var1);
        tcl.evalf("puts \"      my_var2 = %f\"", my_var2);
}
```

Figure 21. Execute OTcl command from a C++ Object

To execute an OTcl command from C++, you should get a reference to "Tcl::instance()" that is declared as a static member variable. This offers you a couple of functions with which you can pass an OTcl command to the interpreter (the first line of "MyPrivFunc" does this). This example shows two ways to

pass an OTcl command to the interpreter. For a complete list of OTcl command passing functions, refer to the NS documentation.

- **Compile, run and test "MyAgent"**

Until now, we examined the essential OTcl linkages available in NS using the "MyAgent" example. Assuming that running and testing this example would help the reader's understanding further, we present a procedure that helps you to compile, run and test "MyAgent".

1. Download "ex-linkage.cc" file, and save it under the "ns-2" directory.
2. Open "Makefile", add "ex-linkage.o" at the end of object file list.
3. Re-compile NS using the "make" command.
4. Download the "ex-linkage.tcl" file that contains "MyAgent" testing OTcl commands. (see Figure 22 for the input script and the result)
5. Run the OTcl script using command "ns ex-linkage.tcl".

ex-linkage.tcl

```
# Create MyAgent (This will give two warning messages that
# no default vaules exist for my_var1_otcl and my_var2_otcl)
set myagent [new Agent/MyAgentOtcl]


# Set configurable parameters of MyAgent
$myagent set my_var1_otcl 2
$myagent set my_var2_otcl 3.14

# Give a command to MyAgent
$myagent call-my-priv-func
```

result

```
warning: no class variable Agent/MyAgentOtcl::my_var1_otcl

        see tcl-object.tcl in tclcl for info about this warning.

warning: no class variable Agent/MyAgentOtcl::my_var2_otcl

Message From MyPrivFunc
     my_var1 = 2
     my_var2 = 3.140000
```

Figure 22. Test OTcl Script and The Result

Add New Application and Agent

(work with ns-2.1b8a)

- **Objective**

We want to build a multimedia application that runs over a UDP connection, which simulates the behavior of an imaginary multimedia application that implements "five rate media scaling" that can respond to network congestion to some extent by changing encoding and transmission policy pairs associated with scale parameter values.

- **Application Description**

In this implementation, it is assumed that when a connection is established, the sender and the receiver agree on 5 different sets of encoding and transmission policy pairs, and associate them with 5 scale values (0-4). For simplicity reasons, it is also assumed that a constant transmission rate is determined for each of the encoding and transmission policy pairs, and every pair uses packets of the same size regardless of the encoding scheme.

Basically, "five rate media scaling" works as follow. The sender starts with transmission rate associated with scale 0, and changes transmission rates according to the scale value that the receiver notifies. The receiver is responsible for monitoring network congestion and determining the scale factor. For congestion monitoring, a simple periodical (for every RTT second) packet loss monitoring is used, and even a single packet loss for each period is regarded as network congestion. If congestion is detected, the receiver reduces the scale parameter value to half and notifies the sender of this value. If no packet loss is detected, the receiver increases the value by one and notifies the sender.

- **Problem Analysis**

Before implementing this application, the UDP agent implementation is examined and one major problem is found. Since a UDP agent allocates and sends network packets, all the information needed for application level communication should be handed to the UDP agent as a data stream. However, the UDP implementation allocates packets that only have a header stack. Therefore, we need to modify the UDP implementation to add a mechanism to send the data received from the application. It is also noted that we might want to use this application for further research on IP router queue management mechanisms. Therefore, we need a way to distinguish this type of multimedia stream from other types of streams. That is, we also need to modify UDP agent to record data type in one of IP header fields that is currently not used.

- **Design and Implementation**

For the application, we decided to take the CBR implementation and modify it to have the "five level media scaling" feature. We examined the C++ class hierarchy, and decided to name the class of this application as "MmApp" and implement as a child class of "Application". The matching OTcl hierarchy name is "Application/MmApp". The sender and receiver behavior of the application is implemented together in "MmApp". For the modified UDP agent that supports "MmApp", we decided to name it "UdpMmAgent" and implement it as a child class of "UdpAgent". The matching OTcl hierarchy name is "Agent/UDP/UDPmm"

- **MmApp Header:** For the application level communication, we decided to define a header of which the structure named in C++ "hdr_mm". Whenever the application has information to transmit, it will hand it to "UdpMmAgent" in the "hdr_mm" structure format. Then, "UdpMmAgent" allocates one or more packets (depending on the simulated data packet size) and writes the data to the multimedia header of each packet. Figure 23 shows the header definition, in which a structure for the header and a header class object, "MultimediaHeaderClass" that should be derived from "PacketHeaderCalss" is defined. In defining this class, the OTcl name for the header ("PacketHeader/Multimedia") and the size of the header structure defined are presented. Notice that bind_offset() must be called in the constructor of this class.

```
// Multimedia Header Structure
struct hdr_mm {
        int ack;      // is it ack packet?
        int seq;      // mm sequence number
        int nbytes;   // bytes for mm pkt
        double time;  // current time
        int scale;    // scale (0-4) associated with data rates

        // Packet header access functions
        static int offset_;
        inline static int& offset() { return offset_; }
        inline static hdr_mm* access(const Packet* p) {
                return (hdr_mm*) p->access(offset_);
        }
};
```

```
// Multimedia Header Class
static class MultimediaHeaderClass : public PacketHeaderClass {
public:
 MultimediaHeaderClass() : PacketHeaderClass("PacketHeader/Multimedia",
                                              sizeof(hdr_mm)) {
            bind_offset(&hdr_mm::offset_);
        }
} class_mmhdr;
```

Figure 23. MM Header Structure & Class (in "udp-mm.h" & "udp-mm.cc")

```
enum packet_t {
        PT_TCP,
        ...
        PT_Multimedia,
        PT_NTYPE // This MUST be the LAST one
};

class p_info {
public:
        p_info() {
                name_[PT_TCP]= "tcp";
                ...
                name_[PT_Multimedia] = "Multimedia";
                name_[PT_NTYPE]= "undefined";
        }
        ...
};
```

Figure 24 (a). Add to the "packet.h" file (C++)

```
foreach prot {
        AODV
        ...
        Multimedia
} {
        add-packet-header $prot
}
```

Figure 24 (b). Add to the "ns-packet.tcl" file (Otcl)

We also add lines to packets.h and ns-packet.tcl as shown in Figure 24 (a) and (b) to add our "Mulltimedia" header to the header stack. At this point, the header creation process is finished, and "UdpMmAgent" will access the new header via hdr_mm::access() member function. Please refer to NS Manual for detailed information on header creation and access methods. For the rest of the application and the modified UDP agent description, refer directly to "mm-app.h", "mm-app.cc", "udp-mm.h" and "udp-mm.cc" files as needed.

- **MmApp Sender:** The sender uses a timer for scheduling the next application data packet transmission. We defined the "SendTimer" class derived from the "TimerHandler" class, and wrote its "expire()" member function to call "send_mm_pkt()" member function of the "MmApp" object. Then, we included an instance of this object as a private member of "MmApp" object referred to as "snd_timer_". Figure 25 shows the "SendTimer" implementation.

```
class SendTimer : public TimerHandler {
 public:
        SendTimer(MmApp* t) : TimerHandler(), t_(t) {}
        inline virtual void expire(Event*);
 protected:
        MmApp* t_;
};

void SendTimer::expire(Event*)
{
  t_->send_mm_pkt();
}
```

```
class MmApp : public Application {
public:
        MmApp();
        ...
private:
        ...
        SendTimer snd_timer_;
        ...
};
```

```
MmApp::MmApp() : running_(0), snd_timer_(this), ack_timer_(this)
{
  bind_bw("rate0_", &rate[0]);
  ...
  bind_bw("rate4_", &rate[4]);
  bind("pktsize_", &pktsize_);
  bind_bool("random_", &random_);
}
```

```
void MmApp::send_mm_pkt()
{
  hdr_mm mh_buf;

  if (running_) {
    ...
    agent_->sendmsg(pktsize_, (char*) &mh_buf);  // send to UDP

    // Reschedule the send_pkt timer
    double next_time_ = next_snd_time();
    if(next_time_ > 0) snd_timer_.resched(next_time_);
  }
}
```

Figure 25. SendTimer Implementation.

Before setting this timer, "MmApp" re-calculates the next transmission time using the transmission rate associated with the current scale value and the size of application data packet that is given in the input simulation script (or use the default size). The "MmApp" sender, when an ACK application packet arrives from the receiver side, updates the scale parameter.

- **MmApp Receiver:** The receiver uses a timer, named "ack_timer_", to schedule the next application ACK packet transmission of which the interval is equal to the mean RTT. When receiving an application data packet from the sender, the receiver counts the number of received packets and also counts the number of lost packets using the packet sequence number. When the "ack_timer_" expires, it invokes the "send_ack_pkt" member function of "MmApp", which adjusts the scale value looking at the received and lost packet accounting information, resets the received and lost count to 0, and sends an ACK packet with the adjusted scale value. Note that the receiver doesn't have any connection establishment or closing methods. Therefore, starting from the first packet arrival, the receiver periodically sends ACK packets and never stops (this is a bug).

- **UdpMmAgent:** The "UdpMmAgent" is modified from the "UdpAgent" to have the following additional features: (1) writing to the sending data packet MM header the information received from a "MmApp" (or reading information from the received data packet MM header and handing it to the "MmApp"), (2) segmentation and re-assembly ("UdpAgent" only implements segmentation), and (3) setting the priority bit (IPv6) to 15 (max priority) for the "MmApp" packets.

- **Modify "agent.h":** To make the new application and agent running with your NS distribution, you should add two methods to "Agent" class as public. In the "command" member function of the "MmApp" class, there defined an "attach-agent" OTcl command. When this command is received from OTcl, "MmApp" tries to attach itself to the underlying agent. Before the attachment, it invokes the "supportMM()" member function of the underlying agent to check if the underlying agent support for multimedia transmission (i.e. can it carry data from application to application), and invokes "enableMM()" if it does. Even though these two member functions are defined in the "UdpMmAgnet" class, it is not defined in its base ancestor "Agent" class, and the two member functions of the general "Agent" class are called. Therefore, when trying to compile the code, it will give an error message. Inserting the two methods as public member functions of the "Agent" class (in "agent.h") as follows will solve this problem.

```
class Agent : public Connector {
 public:
        Agent(int pktType);
        ...
        virtual int supportMM() { return 0; }
        virtual void enableMM() {}
        virtual void sendmsg(int nbytes, const char *flags = 0);
        virtual void send(int nbytes) { sendmsg(nbytes); }
        ...
}
```

Figure 26 (a). Add two member functions to "Agent" class.

- **Modify "app.h":** You also need to add an additional member function "recv_msg(int nbytes, const char *msg)" to "Application" class as shown in Figure 26 (b). This member function, which was included in the "Application" class in the old versions of NS (ns-2.1b4a for sure), is removed from the class in the latest versions (ns-2.1b8a for sure). Our multimedia application was initially written for the ns-2.1.b4a, and therefore requires "Application::recv_msg()" member function for ns-2.1b8a or later versions.

```
class Application : public Process {
public:
        Application();
        virtual void send(int nbytes);
        virtual void recv(int nbytes);
        virtual void recv_msg(int nbytes, const char *msg = 0){};
        virtual void resume();
        ...
};
```

Figure 26 (b). Add a member function to "Application" class.

- **Set default values for new parameter in the "ns-default.tcl":** After implementing all the parts of the application and agent, the last thing to do is to set default values for the newly introduced configurable parameters in the "ns-default.tcl" file. Figure 26 shows an example of setting the default values for configurable parameters introduced by "MmApp".

```
...
Application/MmApp set rate0_ 0.3mb
Application/MmApp set rate1_ 0.6mb
Application/MmApp set rate2_ 0.9mb
Application/MmApp set rate3_ 1.2mb
Application/MmApp set rate4_ 1.5mb

Application/MmApp set pktsize_ 1000
Application/MmApp set random_ false
...
```

Figure 27. Default parameter value settings

- **Download and Compile**

Here is a checklist that should be done before recompiling your NS.

1. Download "mm-app.h", "mm-app.cc", "udp-mm.h" and "udp-mm.cc") to the "ns-2" directory.
2. Make sure you registered the new application header by modifying "packet.h" and "ns-packet.tcl" as shown in Figure 24 (a) and (b).
3. Add supportMM() and enableMM() methods to the "Agent" class in "agent.h" as shown in Figure 26 (a).
4. Add recv_msg() method to the "Application" class in "app.h" as shown in Figure 26 (b).
5. Set default values for the newly introduced configurable parameters in "ns-default.tcl" as described in Figure 27.

After you've done all things in the checklist, modify your "Makefile" as needed (include "mm-app.o" and "udp-mm.o" in the object file list) and re-compile your NS. It is generally a good practice to do "make depend" after you make changes in "Makefile" before a re-compile.

- **Test Simulation**

Figure 28 shows a simulation topology and scenario that is used to test "MmApp", and Figure 29 shows the test simulation script. Download this script and test the newly added components.

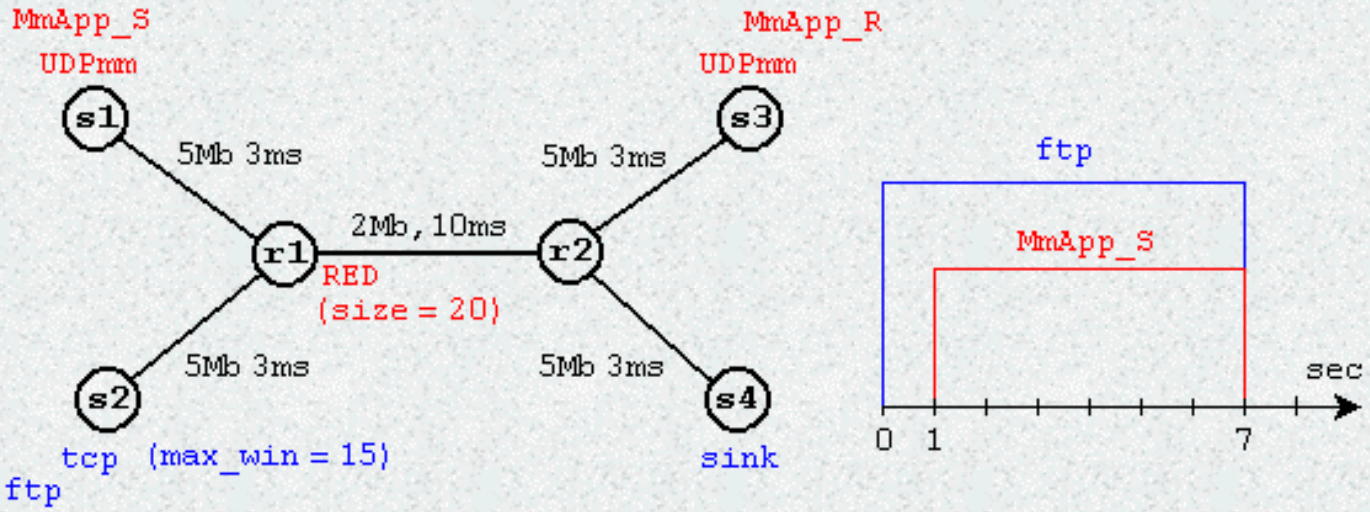

Figure 28. Test Simulation Topology and Scenario

```
set ns [new Simulator]
...
$ns duplex-link $node_(r1) $node_(r2) 2Mb 10ms RED
...

#Setup RED queue parameter
$ns queue-limit $node_(r1) $node_(r2) 20
Queue/RED set thresh_ 5
Queue/RED set maxthresh_ 10
Queue/RED set q_weight_ 0.002
Queue/RED set ave_ 0
...
#Setup a MM UDP connection
set udp_s [new Agent/UDP/UDPmm]
set udp_r [new Agent/UDP/UDPmm]
$ns attach-agent $node_(s1) $udp_s
$ns attach-agent $node_(s3) $udp_r
$ns connect $udp_s $udp_r
$udp_s set packetSize_ 1000
$udp_r set packetSize_ 1000
$udp_s set fid_ 1
$udp_r set fid_ 1

#Setup a MM Application
set mmapp_s [new Application/MmApp]
set mmapp_r [new Application/MmApp]
$mmapp_s attach-agent $udp_s
$mmapp_r attach-agent $udp_r
$mmapp_s set pktsize_ 1000
```

```
$mmapp_s set random_ false
...
#Simulation Scenario
$ns at 0.0 "$ftp start"
$ns at 1.0 "$mmapp_s start"
$ns at 7.0 "finish"

$ns run
```

Figure 29. "MmApp" Test Simulation Script

Add New Queue

(work with ns-2.1b8a)

- **Objective**

To build a simple drop-tail router output queue that uses a round-robin dequeue scheduling for priority 15 packets (from a "MmApp" over "UDPmm") and the other packets in the queue. That is, while priority 15 packets and other packets coexist in the queue, it dequeues the oldest packets of each type one by one in turn.

- **Design**

The queue has two logical FIFO queues, say LQ1 and LQ2, of which the total size is equal to the size of physical queue (PQ), i.e. LQ1 + LQ2 = PQ. To implement a normal drop-tail enqueue behavior, when a packet is to be enqueued, the enqueue manager checks if size of LQ1 + LQ2 is less than maximum allowed PQ size. If so, the packet will be enqueued to an appropriate logical queue. To implement the round-robin dequeue scheduling, the dequeue manager tries to dequeue one packet from a logical queue and the next one from the other logical queue in turn. I.e. packets in the two logical queues are dequeued at 1:1 ratio if both queues have packets.

- **Implementation**

We named the C++ name for this queue object "DtRrQueue" (Drop-Tail Round-Robin Queue) that is derived from "Queue" class. The matching OTcl name is "Queue/DTRR". When the "recv" member function that is implemented in the "Queue" class (in "queue.cc") receives a packet, it invokes the "enqueue" member function of the queue object and invokes "dequeue" if the link object is not blocked. When the link came from a blocked state, it also calls the "dequeue" member function of its queue object. Therefore, we needed to write "enqueue" and "dequeue" member functions of the "DtRrQueue" object class. Figure 30 shows the "DtRrQueue" class definition and its "enqueue" and "dequeue" member functions. For the complete implementation, refer to "dtrr-queue.h" and "dtrr-queue.cc" files. Since the codes are really easy to understand, no further explanation is given.

```
class DtRrQueue : public Queue {
 public:
        DtRrQueue() {
                q1_ = new PacketQueue;
                q2_ = new PacketQueue;
                pq_ = q1_;
                deq_turn_ = 1;
        }
```

```
 protected:
         void enque(Packet*);
         Packet* deque();

         PacketQueue *q1_;    // First  FIFO queue
         PacketQueue *q2_;    // Second FIFO queue
         int deq_turn_;       // 1 for First queue 2 for Second
};
```

```
void DtRrQueue::enque(Packet* p)
{
  hdr_ip* iph = hdr_ip::access(p);

  // if IPv6 priority = 15 enqueue to queue1
  if (iph->prio_ == 15) {
    q1_->enque(p);
    if ((q1_->length() + q2_->length()) > qlim_) {
      q1_->remove(p);
      drop(p);
    }
  }
  else {
    q2_->enque(p);
    if ((q1_->length() + q2_->length()) > qlim_) {
      q2_->remove(p);
      drop(p);
    }
  }
}
```

```
Packet* DtRrQueue::deque()
{
  Packet *p;

  if (deq_turn_ == 1) {
        p =  q1_->deque();
        if (p == 0) {
            p = q2_->deque();
            deq_turn_ = 1;
        }
        else
            deq_turn_ = 2;
  }
  else {
        p =  q2_->deque();
        if (p == 0) {
```

```
            p = q1_->deque();
            deq_turn_ = 2;
        }
        else
            deq_turn_ = 1;
    }

    return (p);
}
```

Figure 30. "DtRrQueue" class implementation

- **Test Simulation**

We used the simulation script used for testing "MmApp" over "UDPmm" in the previous section with changing the RED queue to DTRR queue for the link r1-r2. The changes to the script are show in Figure 31. Download this script and test your newly added queue components.

```
Set ns [new Simulator]

...

$ns duplex-link $node_(s1) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(s2) $node_(r1) 5Mb 3ms DropTail
$ns duplex-link $node_(r1) $node_(r2) 2Mb 10ms DTRR
$ns duplex-link $node_(s3) $node_(r2) 5Mb 3ms DropTail
$ns duplex-link $node_(s4) $node_(r2) 5Mb 3ms DropTail

#Set DTRR queue size to 20
$ns queue-limit $node_(r1) $node_(r2) 20

...

#Simulation Scenario
$ns at 0.0 "$ftp start"
$ns at 1.0 "$mmapp_s start"
$ns at 7.0 "finish"

$ns run
```

Figure 31. "DtRrQueue" test simulation script

LAN Example

(Obsolete: work with ns-2.1b4a)

This section contains a pointer to an example LAN simulation script and shows its network topology and simulation scenario. Download "ex-lan.tcl" file that is modified a little bit from "ns-2/tcl/ex/lantest.tcl" and examine it yourself.

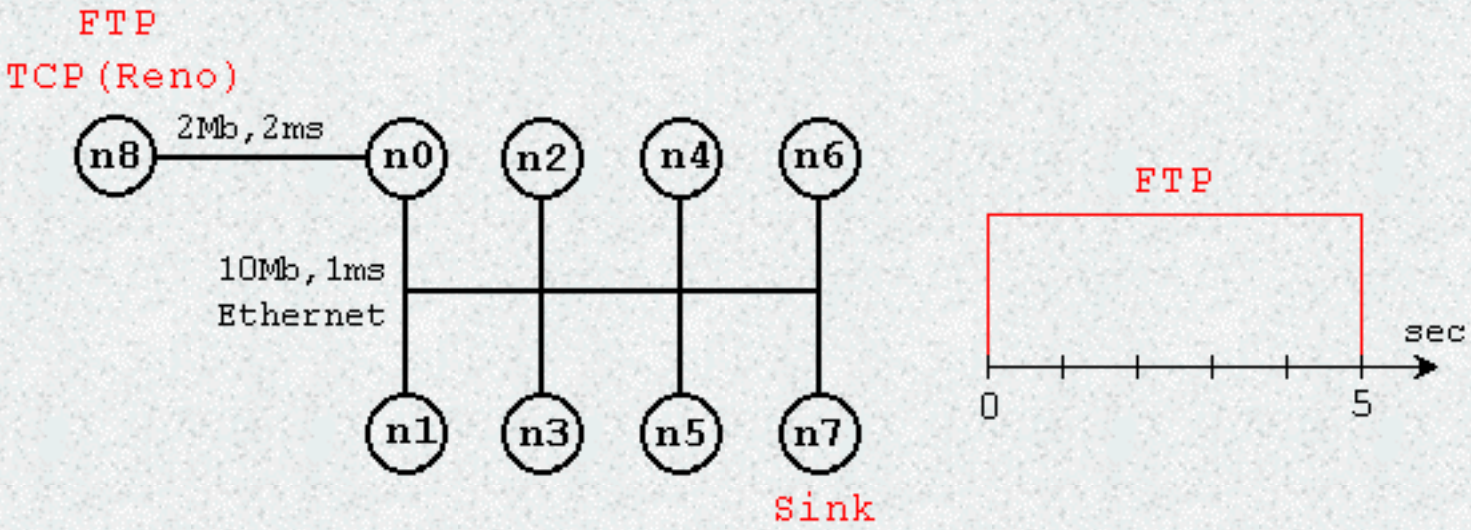

Figure 32. LAN simulation network topology and scenario

Multicasting Example

(Obsolete: work with ns-2.1b4a)

This section contains a pointer to an example multicasting simulation script and shows the NAM screen capture of the simulation. Download "ex-mcast.tcl" file and examine it yourself. This example is from the 5th VINT/NS Simulator Tutorial/Workshop.

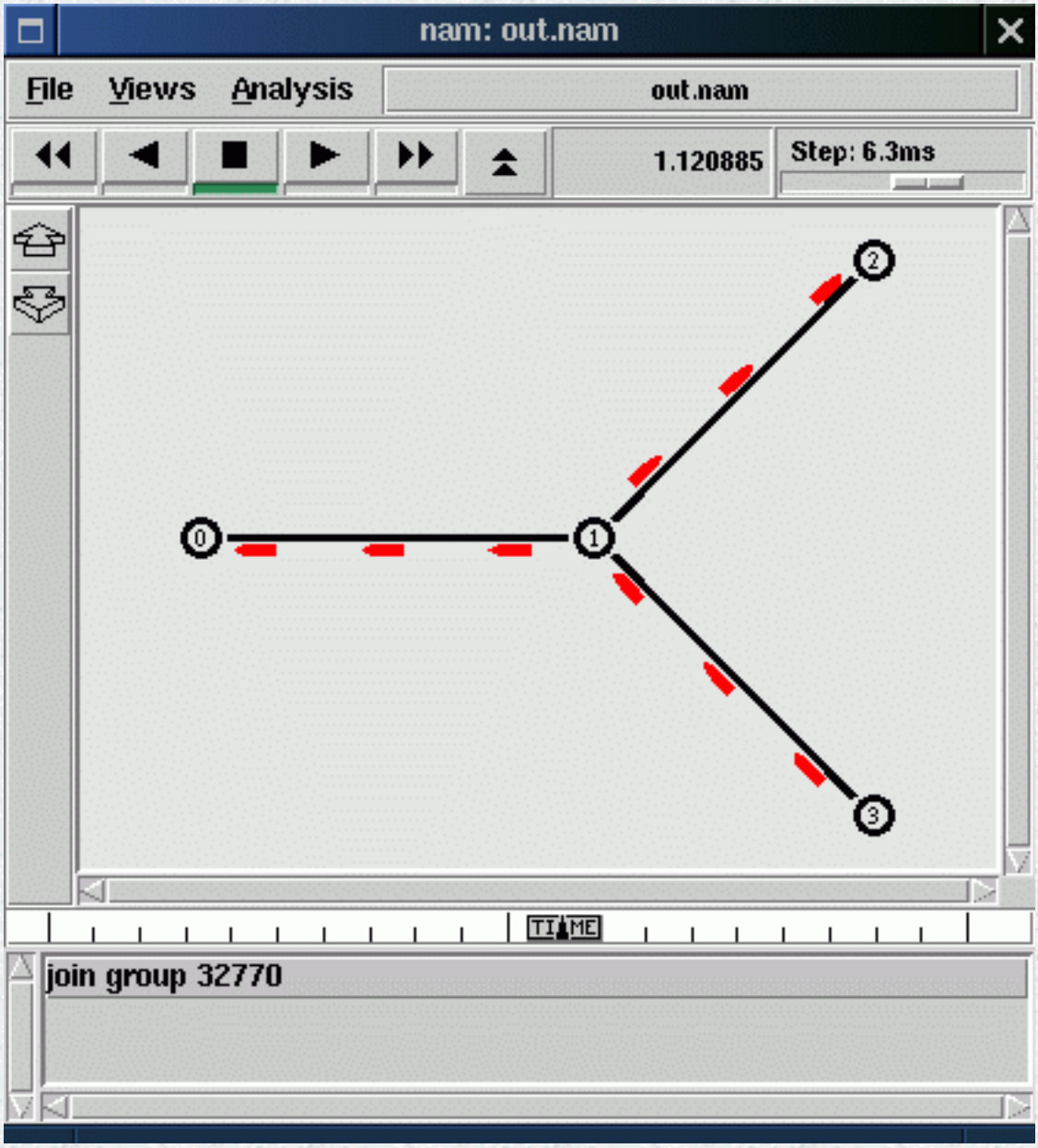

Figure 33. The NAM screen capture of the multicasting simulation

Web Server Example

(Obsolete: work with ns-2.1b4a)

This section contains a pointer to an example Web Server simulation script and shows its network topology. Download "ex-web.tcl" and "dumbbell.tcl" files and examine them yourself. This example is from the 5th VINT/NS Simulator Tutorial/Workshop. Note that you should change the path for "http-mod.tcl" at the begining of the script to an appropriate one to make it run on your machine.

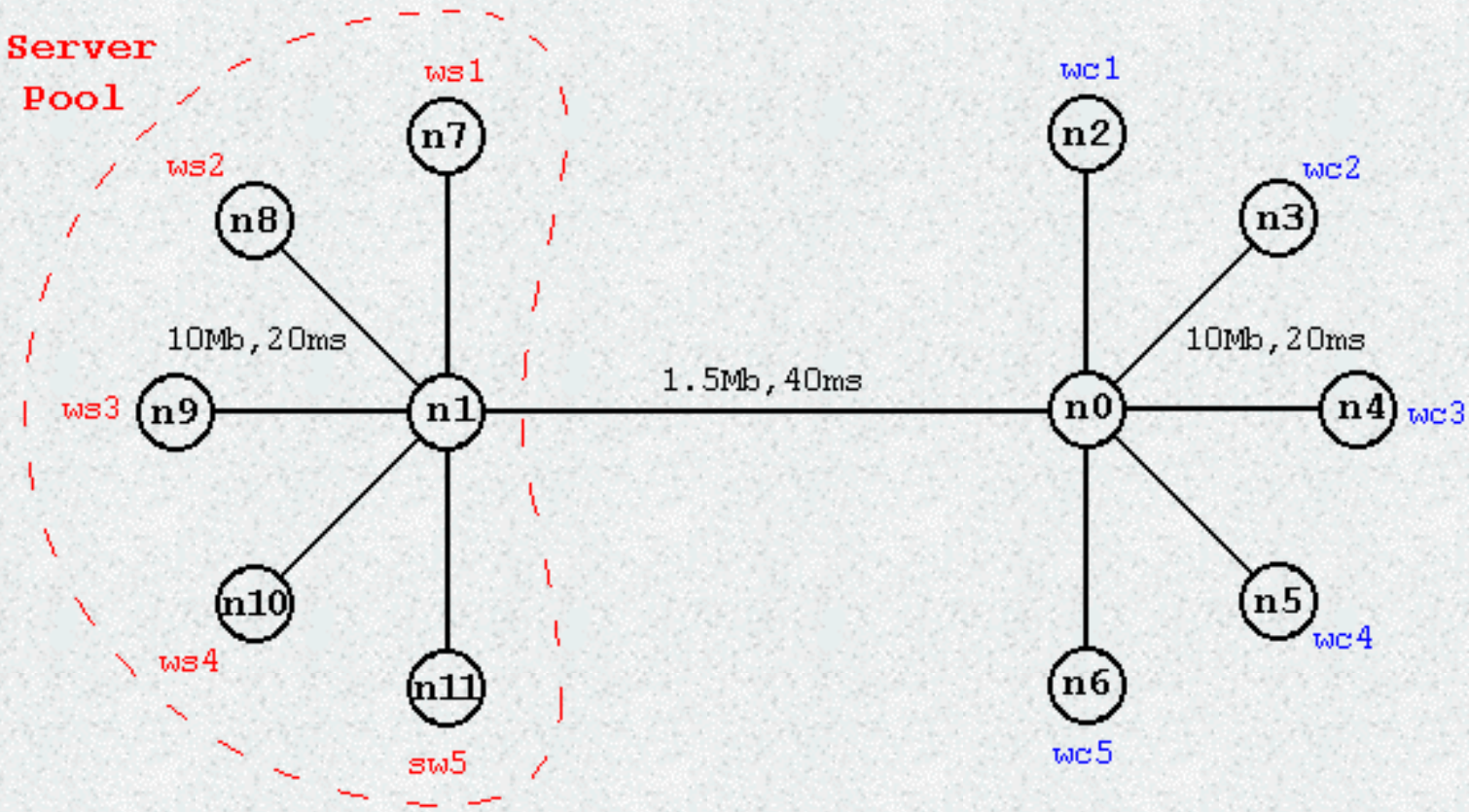

Figure 34. Web server simulation network topology

SRM using NS-2

This section gives an overview of the implementation of SRM in NS. Running an SRM simulation requires

- creating and configuring the agent,
- attaching an application level data source (a traffic generator), and
- starting the agent and the traffic generator.

The key steps in configuring a virgin SRM agent are to assign its multicast group and attach it to a node. Other useful configuration parameters are to assign a separate flow id to traffic originating from this agent, to open log file for statistics, and a trace file for trace data. The agent does not generate any application data on its own; instead the simulation user can connect any traffic generation module to any SRM agent to generate data. The user can attach any traffic generator to n SRM agent. The SRM agent will add the SRM headers, set the destination address to the multicast group and deliver the packet to its target. SRM header contains the type of message, identity of the sender, the sequence number of the message, the round for which this message is being sent.Each data unit in SRM is identified as < *sender's id, message sequence number* >. The SRM agent does not generate its own data; it does not also keep track of the data sent except to record the sequence numbers of messages received in the event that it has to do error recovery. Since the agent has no actual record of the past data, it needs to know what packet size to use for each repair message. The agent and the traffic generator must be started separately using : *$srm start* and *$exp0 start*

At start, the agent joins the multicast group and starts generating the session messages.

Each agent tracks two sets of statistics: statistics to measure the response to data loss and overall statistics for each request/repair.

Data Loss:

The statistics to measure the response to data losses track the duplicate request (and repair), and the average request (and repair) delay. Each new request (or repair) starts a new request (or repair) period. During the request (or repair) period, the agent measures the number of first round duplicate requests (or repair) until the round terminates either due to receiving a request (or repair) or due to the agent sending one.

Overall Statistics:

In addition, each loss recovery and session object keeps a track of times and statistics. In particular, each object records its *startTime. ServiceTime, distance*, are relevant to the object; startTime is the time that this object was created, serviceTime is the time for this object to complete its task, and the distance is the one-way time to reach the remote peer.

For request objects, startTime is the time a packet loss is detected, serviceTime is the time to finally

receive that packet and the distance is the distance to the original sender of the packet. For repair objects, startTime is the time that a request for retransmission is received, serviceTime is the time to send a repair and the distance is the distance to the original requester. For both types of objects, the serviceTime is normalized by distance. For the session object, startTime is the time that the agent joins the multicast group, serviceTime and distance are not relevant.

Each object also maintains statistics particular to that type of object. Request objects track the number of duplicate requests and repairs received, the number of requests sent, and the number of times that this object had to backoff before finally receiving the data. Repair objects track the number of duplicate requests and repairs, as well as whether or not this object for this agent sent the repair. Session objects simply record the number of session messages sent.

The values of timers and the statistics for each object are written to the log file every time an object completes the error recovery function it was tasked to do.

Tracing :

Each object writes out trace information that can be used to track the progress of the object in its error recovery. Each trace entry is of the form:

<prefix> <tag> <type of entry> <values>

The prefix is as described in the previous section for statistics. The tag is Q for requets objects, P for repair objects, and S for session objects.

Architecture and Internals:

The SRM agent implementation splits the protocol functions into packet handling, loss recovery, and session message activity.

- Packet handling consists of forwarding application data messages, sending and receipt of control messages. These activities are executed by C++ methods.
- Error detection done in C++ due to receipt of error messages. However, loss recovery is entirely done through instance procedures in Otcl.
- The sending and processing of messages is accomplished in C++; the policy about when these messages should be sent is decided by instance procedures in Otcl

Packet Handling*:* Processing received messages.

The recv() method can receive 4 types of messages: data, request, repair and session messages.

Data Packets

The agent does not generate any data messages. The user has to specify an external agent to generate traffic. The recv() method must distinguish between locally originated data, that must be sent to the multicast group, and the data received from the multicast group that must be processed. Therefore, the application agent must set the packet's destination address to zero.

For locally originated data, the agent adds the appropriate SRM headers, sets the destination address to the multicast group and forwards the packet to its target.

On receiving a data message from the group, the recv_data(sender, msgid) will update its state marking message <sender, msgid> received, and possibly trigger requests if it detects losses. In addition, if the message was an older message received out of order, then there must be a pending request or repair that must be cleared. In that case, the compiled object invokes the Otcl instance procedure, recv-data {sender,msgid}.

Currently, there is no provision for receivers to actually receive any application data. The agent does not also store any of the user data. It only generates repair messages of the appropriate size, defined by instance variable packetSize_. However, the agent assumes that any application data is placed in the data portion of the packet, pointed to by

packet->accessdata().

Request Packets :

On receiving a request, recv_rqst(sender, msgid) will check whether it needs to schedule requests for other missing data. If it has received this request before it was aware that the source had generated this data message(i.e the sequence no. of this message is higher than the last known sequence no. of data from this source), then the agent can infer that it is missing this, as well as data from the last known sequence no. onwards; it schedules requests for all of the missing data and returns. On the other hand, if the seuqence no. of the request is less than the last know sequence no. from the source, then the agent can be in one of the three states: a) it does not have this data, and has a requets pending for it, b) it has the data, and has seen an earlier request, upon which it has a repair pending for it, or c) it has the data, and it should instantiate a repair. All of these error recovery mechanisms are done in OTcl; recv_rqst() invokes the instance procedure recv-rqst{sender, msgid, requester} for futher processing.

Repair Packets :

On receiving a repair packet, recv_repr(sender, msgid) will check whether it needs to schedule requests for other missing data. If it has received this repair before it was aware that the source had generated this data message(i.e. the sequence no. of the repair is higher than the last known sequence no. from this source), then the agent can infer that it is missing all data between the last known sequence no. and that on the repair; it schedules requests for all of this data, marks this message as received, and returns. On the other hand, if the sequence no. of the request is lesser than the last known sequence no. from the source, then the agent can be in one of the 3 states; a) it does not have the data. and has a request pending for it, b) it has the data, and has seen an earlier request, upon which it has a repair pending for it, or c) it has the data, and probably scheduled a repair for it at some time; after recovery it holds down its timer (equal to three times its distance to some requester)expired, at which time the pending object was cleared. In this last situation, the agent will simply ignore the repair, for lack of being able to do anything meaningful. All of these error recovery mechanisms are done in OTcl; recv_repr() invokes the instance procedure recv-repr{sender, msgid} to complete the loss recovery phase for the particular message.

Session Packets :

On receiving a session message, the agent updates its sequence numbers for all the active sources, and computes its instantaneous distance to the sending agent if possible. This agent will ignore earlier session messages from a group member, if it has received a later one out of order.

Session message processing is done in recv_sess(). The format of the session message is:<count of tuples

in this message, list of tuples>, where each tuple indicates the <sender id, last sequence no. from the source, time the last session message was received from this sender, time that the message was sent>. The first tuple is the information about the local agent.

It is possible to trivially obtain two flavors of SRM based on whether the agents use probabilistic or deterministic suppression. The default request and repair timer parameter for each SRM agent are

Agent/ SRM set C1_ 2.0

Agent/ SRM set C2_ 2.0

Agent/ SRM set D!_ 1.0

Agent/ SRM set D2_ 1.0

***Deterministic Suppressions*:**

Class Agent/ SRM/ Deterministic – superclass Agent / SRM

Agent/SRM/ deterministic sets C2_ 0.0

Agent /SRM/ deterministic sets D2_ 0.0

***Probabilistic Suppressions*:**

Class Agent/ SRM/ Probabilistic – superclass Agent / SRM

Agent/SRM/ Probabilistic sets C1_ 0.0

Agent /SRM Probabilistic sets D1_ 0.0

Loss Detection:

The SRM agent implementation splits the protocol functions into

- Packet handling,
- Loss recovery, and
- Session message activity.

Packet handling consists of forwarding application data messages, sending and receipt of control messages. These applications are executed by C++ methods.

Error detection is done in C++ due to receipt of messages. However, the loss recovery is entirely done through instance procedures in OTcl.

Sending and processing of messages is accomplished through C++; the policy about when these messages should be sent is decided by instance procedures in OTcl.

A very small encapsulating class, entirely in C++, tracks a number of assorted state information. Each member of a group, ni, uses one SRMinfo block for every other member of the group. An SRMinfo object about group member nj at ni, contains information about session messages received by ni from nj.

Ni can use this information to compute its distance to nj. If nj, sends is active in sending data traffic, then the SRMinfo object will also contain information about the received data, including a bit vector indicating all packets received from nj.

The agent keeps a list of SRMinfo objects, one per group member, in its member variable, sip_. Its method get_state(int sender) will return the object corresponding to that sender, possibly creating that object, if it did not already exist. The class SRMinfo has two methods to access and set the bit vector, i.e.

IfReceived(int id) indicates whether the particular message from the appropriate sender, with id id was received at ni.

SetReceived(int id) to set the bit to indicate that the particlar message from the appropriate sender, with id id was received at ni.

The session message variables to access timing information are public; no encapsulating methods are provided. These are:

```
Int lsess_; /* # of last session msg. Received */
Int sendTime_; /* Time session msg. # sent */
Int recvTime_; /* Time session message # received */
Double distance_; /* Delay between nodes */
int ldata_; /* Data Messages */
```

Loss Recovery Objects:

Timers are used to control when any particular control message is to be sent. The SRM agent uses a separate *class SRM* to do the timer based processing. *Class SRM* is used for sending periodic session messages. An SRM agent will instantiate one object to recover from one lost data packet. Agents that detect the loss will instantiate an object in the *class SRM/request;* agents that receive a request and have the required data will instantiate an object in the *class SRM/ repair.*

Request Mechanisms :

SRM agents detect loss when they receive a message, and infer the loss based on the sequence number on the message received. Since packet reception is handled entirely by the compiled object, loss detection occurs in the C++ methods. Loss recovery, however, is handled by instance procedures of the corresponding object in OTcl.

Repair Mechanisms:

The agent will initiate a repair if it receives a request for a packet, and it does not have a request object pending_ for that packet. The default repair object belongs to the *class SRM/repair*. Barring minor differences, the sequence of events and the instance procedures in this class are identical to those for SRM/request.

Sample Code :

```
# STAR TOPOLOGY
```

```
source /usr/lib/ns-allinone-2.1b4/ns-2/tcl/mcast/srm-nam.tcl ;# to separate $
source /usr/lib/ns-allinone-2.1b4/ns-2/tcl/mcast/srm-debug.tcl ;# to trace del$
Simulator set NumberInterfaces_ 1
set ns [new Simulator]
Simulator set EnableMcast_ 1
$ns trace-all [open out.tr w]
$ns namtrace-all [open out.nam w]
set srmSimType Probabilistic
$ns color 0 red ;#data source
$ns color 40 blue ;#session
$ns color 41 green ;#request
$ns color 42 white ;#repair
$ns color 4 red ;#source node

# Creating The Nodes
set nmax 8
for {set i 0} {$i <= $nmax} {incr i} {
    set n($i) [$ns node]
}
$n(1) color "red"

# Creating The Links
for {set i 1} {$i <= $nmax} {incr i} {
    $ns duplex-link $n($i) $n(0) 1.5Mb 10ms DropTail
}

# Orienting The Links
$ns duplex-link-op $n(0) $n(1) orient right
$ns duplex-link-op $n(0) $n(2) orient right-up
$ns duplex-link-op $n(0) $n(3) orient up
$ns duplex-link-op $n(0) $n(4) orient left-up
$ns duplex-link-op $n(0) $n(5) orient left
$ns duplex-link-op $n(0) $n(6) orient left-down
$ns duplex-link-op $n(0) $n(7) orient down
$ns duplex-link-op $n(0) $n(8) orient right-down
set group 0x8000
set cmc [$ns mrtproto CtrMcast {}]
$ns at 0.3 "$cmc switch-treetype $group"

# SRM TRACE EVENTS
set srmStats [open srmStats.tr w]
set srmEvents [open srmEvents.tr w]
set fid 0
for {set i 1} {$i <= $nmax} {incr i} {
    set srm($i) [new Agent/SRM/$srmSimType]
    $srm($i) set dst_ $group
```

```
    $srm($i) set fid_ [incr fid]
    $srm($i) log $srmStats
    $srm($i) trace $srmEvents
    $ns at 0.5 "$srm($i) start"
    $ns attach-agent $n($i) $srm($i)
}

# Attach a CBR Agent to srm(1)
set packetSize 800
set s [new Application/Traffic/CBR]
$s set packet_size_ $packetSize
$s set interval_ 0.02
$s attach-agent $srm(1)
$srm(1) set tg_ $s
$srm(1) set app_fid_ 0
$srm(1) set packetSize_ $packetSize
$ns at 2.0 "$srm(1) start-source"
set loss_module [new SRMErrorModel]
$loss_module drop-packet 2 10 1
$loss_module drop-target [$ns set nullAgent_]
$ns at 0.75 "$ns lossmodel $loss_module $n(1) $n(0)"
$ns at 4.0 "finish $s" proc distDump interval {
    global ns srm
    foreach i [array names srm] {
        set dist [$srm($i) distances?]
        if {$dist != ""} {
            puts "[format %7.4f [$ns now]] distances $dist"
        }
    }
    $ns at [expr [$ns now] + $interval] "distDump $interval"
}
proc finish src {
    global prog ns env srmStats srmEvents srm nmax
    $src stop
    $ns flush-trace
    close $srmStats
    close $srmEvents
    puts "converting output to nam format..."
    if [info exists env(DISPLAY)] {
        puts "running nam..."
        exec nam out.nam &
    } else {
        exec cat srmStats.tr >@stdout
    }
    exit 0
}
```

$ns run

WPI WORCESTER POLYTECHNIC INSTITUTE

Computer Science

NS@WPI FAQ

---

1. **Is there an NS tutorial somplace?**

   Try: http://www.isi.edu/nsnam/ns/tutorial/

---

2. **How do setup to use NS on nile?**

   Add `/users/ns/bin` is in your path:

   ```
   set path=(/users/ns/bin $path) # for csh/tcsh
   ```

   Make a working directory for your scripts:

   ```
   cd /users/claypool
   mkdir my-sim
   ```

   Then, make sims and run normally.

---

3. **How do setup to modify NS (edit the simulator) on nile?**

   Check that you are part of the "ns" group:

   ```
   groups (if you don't see "ns" send email to claypool@cs)
   ```

   Set your `CVSROOT` variable (for tcsh):

   ```
   setenv CVSROOT /users/cvsroot (put in your .tcshrc)
   ```

   Make a working directory:

   ```
   cd /users/claypool
   mkdir my-ns
   ```

   Check out ns-2:

   ```
   cd /users/claypool/my-ns
   cvs co ns-2
   ```

   Link in supporting directories:

```
cd /users/claypool/my-ns
ln -s /users/ns/* .
(will give error for ns-2 link, but that's ok)
```

Compile ns:

```
cd /users/claypool/my-ns/ns-2
make
```

Make sure your scripts now use your NEW ns under `/users/claypool/my-ns/ns-2/ns`! Then, make sims and run normally.

To edit a file:

```
cd /users/claypool/my-ns/ns-2
(edit file)
(compile, debug, test, compile, debug test...)
(MAKE SURE YOUR CODE COMPILES AND WORKS BEFORE SUBMITTING)
cvs commit
```

To add a file:

```
cd /users/claypool/my-ns/ns-2
(edit new file)
(compile, debug, test, compile, debug test...)
(MAKE SURE YOUR CODE COMPILES AND WORKS BEFORE SUBMITTING)
cvs add
cvs commit
```

---

4. **Where are some sample scripts I can start from?**

   Many sample scripts can be found in the NS distribution in `/users/ns-2/ns-2/tcl/ex` and `/users/ns/ns-2/tcl/test`. You might also try `/users/ns/ns-2/tcl/ns-tutorial/`.

---

5. **Entering flows by hand is a pain. How do you "generate", say, 200 flows automatically?**

   You can do so with a for loop. Here is an example:

```
#########################
# Creat Network Topology #
#########################

#Set number of TCP connections
set tcp_num 50

#Making two network nodes
set n(1) [$ns node]
set n(2) [$ns node]

#Making edge nodes
```

```
for {set i 1} {$i <= $tcp_num} {incr i} {
    set s($i) [$ns node]
    set r($i) [$ns node]
}

#Creating the network link
$ns duplex-link $n(1) $n(2) 20Mb 20ms DropTail

#Creating edge links
for {set i 1} {$i <= $tcp_num} {incr i} {
    $ns duplex-link $s($i) $n(1) 20Mb 5ms DropTail
    $ns duplex-link $n(2) $r($i) 20Mb 5ms DropTail
}


##########################
# Setup FTP-TCP connections #
##########################

#Setup TCP
for {set i 1} {$i <= $tcp_num} {incr i} {
    set tcp($i) [new Agent/TCP/Reno]
    set sink($i) [new Agent/TCPSink]
    $ns attach-agent $s($i) $tcp($i)
    $ns attach-agent $r($i) $sink($i)
    $ns connect $tcp($i) $sink($i)

    $tcp($i) set ecn_ true
    $tcp($i) set fid_ $i
    $tcp($i) set window_ 20
    $tcp($i) set packetSize_ 1000
}

#Setup FTP Applications
for {set i 1} {$i <= $tcp_num} {incr i} {
    set ftp($i) [new Application/FTP]
    $ftp($i) attach-agent $tcp($i)
    $ftp($i) set type_ FTP
}

##########################
# Start FTP Applications #
##########################
for {set i 1} {$i <= $tcp_num} {incr i} {
    $ns at 0 "$ftp($i) start"
    $ns at 80 "$ftp($i) stop"
}

$ns run
```

---

6. **How do I measure things like packet loss, throughput, etc?**

You pull data you need from the trace file NS produces. That file is in text format, so you can parse it by whatever tools you would like. For example, to calculate throughput you'd look for how many packets arrived per second at a node coming from another node maybe with the protocol TCP.

---

7. **Where are the defaults kept for the NS objects?**

   `/users/ns/ns-2/tcl/lib`

---

8. **Are there some tools somewhere that do some things I would like?**

   Not as many as we'd like. But there is a start in: `/users/ns/tools`

---

9. **Does NS simulate the 3-way TCP handshake?**

   Yes. You set the `syn_` variable to TRUE in `Agent/TCP` in order to simulate the 3-way handshake. In ns-2.1b7a, syn_ is set to FALSE by default. However since ns-2.1b8, the default setting for syn_ has been changed to TRUE.

---

10. **I get the exact same drops every time with my RED router. Is there a way to "seed" the NS random number generator?**

    You do it with the RNG class from OTcl. For example, a new RNG is created and seeded with:

```
set rng [new RNG]
$rng seed 0 # seeds the RNG heuristically;
$rng seed n # seeds the RNG with value n;
```

    Other uses could be:

```
$rng next-random #  return the next random number;
$rng uniform a b # return a number uniformly distributed on [a, b];
$rng integer k   # return an integer uniformly distributed on [0, (k-1)];
$rng exponential # return a number from an exponential distribution
                 # with average 1.;
```

    Currently there is no way to select predefined seeds from OTcl.

Hot Links

- Congestion Control Research Group: http://perform.wpi.edu/cc/
- VINT Project Home Page: http://www.isi.edu/nsnam/vint/
- NS Home Page: http://www.isi.edu/nsnam/ns/
- NS Installation: http://www.isi.edu/nsnam/ns/ns-build.html
- NS Documentation: http://www.isi.edu/nsnam/ns/ns-documentation.html
- NS Manual Page (very out of date): http://www.isi.edu/nsnam/ns/ns-man.html
- NS CVS history (directory structure): http://www.isi.edu/cgi-bin/nsnam/cvsweb
- NS Class Hierarchy: http://www-sop.inria.fr/rodeo/personnel/Antoine.Clerget/ns
- 5th VINT/NS Simulator Tutorial/Workshop: http://www.isi.edu/nsnam/ns/ns-tutorial/ucb-tutorial.html
- Tcl/Tk Quick Reference Guide: http://www.slac.stanford.edu/~raines/tkref.html
- OTcl Tutorial (Berkeley Version): http://bmrc.berkeley.edu/research/cmt/cmtdoc/otcl
- OTcl Tutorial (MIT Version): ftp://ftp.tns.lcs.mit.edu/pub/otcl/README.html
- Guide to awk: http://www.canberra.edu.au/~sam/whp/awk-guide.html
- GNU Plot: http://shazam.econ.ubc.ca/gnuplot.html
- XGraph: http://jean-luc.ncsa.uiuc.edu/Codes/xgraph
- Network Animator (NAM): http://www.isi.edu/nsnam/nam/