User Manual For NS with TCP/IP Over ATM

Simulation Capability

This manual provides a tutorial introduction about how to use NS for simulating networks. It focuses on the general usage and the input commands available to the user. If you wish to modify NS, you should then also read the NS Programmers Manual. Our description of the usage of NS commands is limited largely because we did not have any communication with the actual authors of this program. For those who have no prior exprience with TCL, should read the manpages of ***Tcl*** and ***tclsh.*** Without this the true power of the Tcl Script may not be realised.

History Of NS (Network Simulator)

Work on the LBL Network Simulator began in May 1990 with modifications to S. Keshav's (keshav@research.att.com) REAL network simulator, which he developed for his Ph.D. work at U.C. Berkeley. In Summer 1991, the simulation description language was revamped, and later, the NEST threads model was replaced with an event driven framework and an efficient scheduler. Among other contributions, Sugih Jamin (jamin@usc.edu) contributed the calendar-queue based scheduling code to this version of the program, which was known as tcpsim. In December 1994, McCanne ported tcpsim to C++ and replaced the yacc-based simulation description language with a Tcl interface, and added preliminary multicast support. Also at this time, the name changed from tcpsim to the more generic ns. Throughout, Floyd has made modifications to the TCP code and added additional source models for her investigations into RED gateways, resource management, class-based queuing, explicit congestion notification, and traffic phase effects. Many of the papers discussing these issues are available through URL http://www-nrg.ee.lbl.gov/ .

In this document we will describe the various features available in the simulator and how they can be used to model real networks. We also provide some sample input files for ease of understanding and use of the simulator.

Brief Description

NS is an event-driven network simulator embedded into the Tool Command Language, Tcl. An extensible simulation enginge is implemented in C++ and is configured and controlled via a Tcl interface. The simulator is invoked via the ns interpreter, which is an extension of the vanilla tclsh(1) command shell. All interaction with the interpreter is via a single, new Tcl procedure, ‘‘ns’’. (The ns procedure is present in the "ns_default.tcl" file)

A simulation is defined by a Tcl program. Using the ns command, a network topology is defined, traffic sources and sinks are configured, statistics are collected, and the simulation is invoked. By building upon a fully functional language, arbitrary actions can be programmed into the configuration. The simulation is run via the ns run command, and continues until there are no more events to be processed. For example, a traffic source might be started at time 0 and stopped at time 10 (using the ns at command). Once all traffic sources are stopped, all pending packets will be eventually delivered and no more events will remain. At this time, the original invocation of the ns run command returns and the Tcl script can exit or invoke another simulation run after possible reconfiguration. Alternatively, the simulation can be prematurely halted by invoking the ns stop command or by exiting the script with Tcl’s standard exit command.

File Organization

The simulator files are arranged in three directories. The docs directory contains the documentation including the online html manual. The src directory contains the source code. The bin directory contains the installed binaries.

Basics: Nodes, Links and Agents

The network topology is specified by using three primitive building blocks: **nodes, links, and agents**. Nodes are created with the ***ns node*** command and arranged in a network topology with the ***ns link*** command. Nodes are passive objects which act as containers for ***agents***, the objects that actively drive the simulation. Each node has an automatically-generated, unique address. Traffic sources and sinks and dynamic routing modules are examples of agents. The ***ns agent*** command is used to create an agent at a certain node. Thus ***nodes*** can be thought of the physical entity representing the computer while the ***agent*** can be thought of as the computer process generating and handling the actual packets. The ns agent, ns node, and ns link commands all create new objects and return a procedure that is used to access the object. Once an object is created, it can be manipulated in an object-oriented fashion using the new procedure. More details wiill follow in the User-Commands section.

User Input

The user can enter the input in two forms. One way is to run "***ns***" and enter the commands line by line or the user can specify a file name on the command line which contains the input data. Any arguments that need to be supplied with the file can be put on the command line. For example, suppose you have defined several procedures in a file. Now all you have to do is to put the procedure name on the command line and at the end of the input file (which only contains procedure definations) you can simply put the line "`$argv`". In this way after defining the procedures whatever procedure name you pass as the command line option will get called and executed. The sample input file, **test-suite.tcl,** has many procedures with names starting with "*test_*". So if you run ns with "*ns test-suite.tcl tahoe1"*, the procedure that will actually get called will be test_tahoe1 because of the following lines at the end of the file.

```
if { $argc != 1 } {
```

```
        puts stderr {usage: ns test-suite.tcl [ tahoe1 tahoe2 ... reno reno2 ... ]}
        exit 1
}
if { "[info procs test_$argv]" != "test_$argv" } {
        puts stderr "test-suite.tcl: no such test: $argv"
}
test_$argv
```

NS Defaults

The default values for most variables are specified in the **ns_default.tcl** file. If you have to change any of the defaults you can change them in this file. However if you do this you will have to recompile the program. A collection of such configuration parameters that can be modified as above either before a simulation begins, or dynamically, while the simulation is in progress. If a parameter is not explicitly set, it defaults to a value stored in a global Tcl array. For example, *ns_tcp(window-size)* contains the default window size for tcp objects. A set of defaults are built into the ns interpreter and can be explicitly overridden by simple assignmet. For example, we can say

```
set ns_tcp(window-size) 30
```

to make all future tcp agent creations default to a window size of 30. These defaults include the TCP traffic defaults like w*indow parameters and packet size,* parameters for the **RED Algorithm** , **Class-Based Queueing** and other algorithms including link defaults. For more details see the **ns_default.tcl** file.

Besides the default parameter values, the ns_default.tcl file also contains some functions. These functions have been provided so as to make it easier for the user to specify the input. These include the following:

ns_duplex { n1 n2 bw delay type }

This function is used for creating a duplex link between the nodes n1 and n2 with the specified bandwidth and delay. The type an be one of the following *drop-tail, lossy-uniform, lossy-det, red, cbq, wrr-cbq or atm.* For example, you create two nodes s1 and s2 using the ***ns node*** command Then you can link them together as shown

```
set s1 [ns node]
set s2 [ns node]
ns_duplex $s1 $s2 8Mb 5ms drop-tail
```

This function returns the two links created. So you store them in a variable and then seperately set their individual parameters as follows where [lindex $L 0] means the first element of the two links.

```
set L [ns_duplex $s1 $s2 800kb 100ms drop-tail]
[lindex $L 0] set queue-limit 6
[lindex $L 1] set queue-limit 6
```

ns_create_connection { srcType srcNode sinkType sinkNode class }

This function is used for creating a connection between a source and a sink node. The types include *tcp, tcp-fack, tcp-reno, tcp-newreno, tcp-sack1, tcp-sink, tcp-sink-da (*delayed ACK)*,* *sack1-tcp-sink* and some others. For a complete list see the **ns agent** command. You can use it as follows:

```
set tcp1 [ns_create_connection tcp $s1 tcp-sink $s2 0]
```

Some functions exist for creating certain special connections. These are just slight variations of the **ns_create_connection** command and include :

ns_create_reno { tcpSrc tcpDst window start class} for creating a reno source connection. It attaches an ftp source with starting time **start.**

ns_create_cbr { srcNode sinkNode pktSize interval class } for creating a cbr connection at node srcNode and connect it to a loss-monitor sink agent at node sinkNode. Also connect the sink agent to the source agent. The traffic class of the source agent is agent. The cbr source sends packet of size pktSize bytes at intervals of interval seconds. This can be used as follows:

```
set cbr [ns_create_cbr $source $dest $pktSize $interval $class ]
```

ns_create_class { parent borrow allot maxIdle minIdle priority depth extraDelay} for creating a class-based queueing class. Sets the allotment, maxidle, minidle, priority, depth, and extradelay parameters to the indicated values. There are some other similar functions also available which the user can use. These include ***ns_create_class1, ns_class_params, ns_class_maxIdle, ns_class_minIdle.***

NS Commands

The Procedure ns contains the following commands:

ns connect $tcpnode $atmnetwork $atmnode

This connects a TCP node with the ATM node specified in the ATM network. This enforces that one TCP node can be connected with only one ATM node in a given ATM network. The way the program is designed, the TCP nodes practically know only about the ATM network but not about the nodes inside. Thus the ATM network must be specified. This command does not return anything.

ns add-node $atmnetwork $atmnode

This command inserts an atmnode into the ATM Network. Since a network topology can contain multiple ATM networks, it is required that the ATM node be attached to an ATM Network.

ns node

This command creates a node in the network. If no further arguments are supplied the node created is a TCP node and further agents can be attached to this node. All the ***ns node*** commands return the node that has been created. The ns node command can be used in the following ways.

ns node atm-network

ns node atm-network $routing_algo

Create an ATM Network with the specified routing logic. If routing algorithm is not specified then the **min-hop** algorithm is used. Right now only the **min-hop** algorithm is implemented. The ATMNetwork is just another node in the TCP network.

ns node atm-switch $type ($discard_algo NOT YET DONE)

Create an ATM Switch with the specified type. Right now the types are **output** and **shared.**

ns node atm-source

Create an ATM Source. It is different from a switch in the sense that it can generate ATM Connections of user specified Qualities of Service.

ns link

ns link $src $dst $type

This command creates a undirectional link from src to dst of the type $type. Links are inherently uni-directional and two unidirectional links can be combined to give a bidirectional link. The type specifies the queuing discipline to be used at the link. The nodes must have been created previously by the **ns node** command. A helper procedure, ns_duplex, is defined to facilitate building point-to-point links. The types include :

- ***drop-tail* -** Drop packets from the tail
- ***red* -** The Random Early Detection Gateway
- ***cbq* -** Class-Based Queueing
- ***wrr-cbq*** - Weighted Round Robin class-based queueing
- ***lossy-uniform* -** A Uniform Lossy link, i.e. loose packets uniformly
- ***lossy-det* -** A Deterministic Lossy Link.
- ***atm* -** An ATM Link. It must be connected with ATM Nodes (switches and sources)

ns link

Return a list of the names of all the link objects that have been created.

ns link node1 node2

Return the name of the link object that connects node1 and node2 (there may be only one such object)

ns agent $type $node

This command is used to create an agent of a certain type and then attaching it to a node. The node must have been previously defined with the the **ns node** command. The types include

- **ivs/source**
- **ivs/receiver**
- **loss-monitor** - a CBR sink that reports losses
- **message**
- **rlm-sender**
- **rlm-receiver**
- **tcp-fack** - BSD Reno TCP with forward ACKs
- **tcp-newreno** - a modified version of BSD Reno TCP
- **tcp-reno** - BSD Reno TCP
- **tcp-sack1** - BSD Reno TCP with selective ACKs
- **tcp-sink** - a standard TCP sink
- **tcp-sink-da** - a TCP sink that generates delayed ACKs
- **sack1-tcp-sink** - a TCP sink that generates selective ACKs
- **sack1-tcp-sink-da** - a delayed-ack TCP sink with selective ACKs
- **cbr** - a constant bit rate traffic source
- **tcp -** BSD Tahoe TCP

This command returns the agent created. Some helper functions are defined in ns_default.tcl to assist in the creation of such connections. For example **ns_create_connection** procedure which internally calls **ns agent**

ns trace

This command creates a new trace object and then returns the object. A trace can be bound either to a file, using
a file identifier created with the standard Tcl open command, or to a Tcl procedure so that arbitrary event logging can be carried out by the user script.

ns at $time "proc"

This command can be used to generate an event at time **$time.** The event should be a valid TCL command that you can put anywhere in your input file and it would work. This command can be used to dynamically reconfigure the simulator, dump statistics at specified intervals, start and stop sources, etc. For example after creating an ftp source we can set its start time by

```
ns at 1.0 "$ftp start"
```

where **start** is a command for any data-source. (See data-source.cc)

ns now

This command returns the current time. It can be used for printing time for generating statistics.

ns random

ns random $seed

This command generates a new random number seed.

ns run

After declaring the complete network topology, use this command to actually run the simulator.

Configuration parameters

NS has some simulation parameters that are usually fixed during the entire simulation (like a link bandwidth), but these can be changed dynamically if desired. In this section we will discuss how to do access these configuration parameters and what configuration parameters are available. Each object also has a set of State variables that are specific to a given object and that object's implementation.

Object Methods

In NS all objects including nodes, links and agents are derived from the TclObject class. Thus all such objects have some common methods available which can be executed on an object by invoking that object as a Tcl procedure call. The object class is specified implicitly by the object variable name in the description. For example, $tcp implies the tcp object class and all of its child classes, while $generic implies any object class.

$generic set var value

Sets the member variable **var** of the generic object to value. This command behaves analogously to the Tcl **set** command, but the variables are private to each object and ***only the predefined configuration parameters can be set*** (i.e., new variables cannot be created).

$generic get var

Return the current value of the parameter var that is private to the generic object. **var** is either a configuration parameter or a state variable of the generic object.

$generic trace trace

Attach the trace identified by trace to the generic object. trace must be an identifier return by the **ns trace** command.

Link Methods

$link reset
Reset all internal statistics variables associated with this link. It is useful if you want to start collecting statistics from a certain time.

$link integral util | qsize | qlen
Return the integral over time of the utilization, queue size in bytes, or queue length in packets,

according to the third argument. For example, the average queue delay at a give link can be computed by dividing the elapsed simulation time into expr "8 * [$link integral qsize] / [$link get bandwidth]" . The advantage of this is that by writing simple Tcl procedures the user can gather statisitcs at his/her own will.

$link stat $class $type
Return a per-class statistic associated with this link. class identifies the traffic class of interest (all packets are assigned an integer class identifier which is determined by the class object variable that sourced the packet). type identifies the statistic of interest and may be one of: ***packets, bytes, drops, mean-qlen, mean-qdelay,*** or ***mean-qsize***. ***packets, bytes,*** and ***drops*** give counts of the number of occurrences of each event for the given traffic class. ***mean-qlen*** gives the average number of packets in the queue on this link as seen by class class. Similarly, mean-qsize gives the average number of bytes in the queue.

$link queue-trace $trace
This attaches the specified trace object to the link. This is useful for tracing the exact times of how the link is behaving. It can be useful when the packet by packet trace of the network is supposed to be done.

$link flow-mgr $flow-manager
This attaches the specified flow manager to be attached to this link

$link install $src $dst
This attaches the link between the source and the destination. A user can not create multiple links between two nodes.

Link Configuration Parameters

These parameters default to the values as specified in ns_default.tcl file.

- **bandwidth** - The bandwidth of the link, in bits per second.
- **delay** - The latency of the link, in seconds.
- **queue-limit** - The maximum number of packets that can be queued at the link.i

State Variables

- **queue-length** - The current number of packets queued at this link.
- **queue-size** - The current number of bytes in all packets queued at this link.

Agent Methods

$agent addr
Return the address of the node to which this agent is attached.

$agent node
Returns the name of the node to which this agent is attached

$agent seqno
Returns the sequence number of the cells sent so far.

$agent port
Return the transport-level port of the agent. Ports are used to identify agents within a node.

$agent dst-addr
Return the address of the node this agent is connected to.

$agent dst-port
Return the port that this agent is connected to.

$agent join $group
Add this agent to the multicast host group identified by the address group. This causes the group membership protocol to arrange for the appropriate multicast traffic to reach this agent.

$agent leave $group
Leave the multiast group.

$agent node $node
Attaches this agent to the specified node

$agent connect addr port
Connect this agent to the agent identified by the address addr and port port. This causes packets transmitted from this agent to contain the address and port indicated, so that such packets are routed to the intended agent. The two agents must be compatible (e.g., a tcp-source/tcp-sink pair as opposed a cbr/tcp-sink pair). Otherwise, the results of the simulation are unpredictable.

Agent Configuration Parameters

class

The traffic class of packets generated from this agent. Traffic classes can be used for aggregating (or separating) flows for sta- tistical data reduction and are also used by the CBQ (class-based queuing) module. For example, in a TCP simulation, you might assign a different traffic class to each connection.

There are no state variables specific to the generic agent class.

Node Methods

$node addr
Return this node's address.

$node agent port
Return the Tcl object name of the agent attached to port port on this node. Returns an empty string if

the port is not in use.

There are no state variables or configuration parameters specific to the node class.

Drop-tail Object

Drop-tail objects are a subclass of link objects that implement simple FIFO queue. There are no methods, configuration parameter, or state variables that are specific to drop-tail objects.

Red Object

RED objects are a subclass of link objects that implement random early-drop queuing. There are no object methods that are specific to RED objects.

Red Configuration Parameters

- **bytes** - Set to "1" to measure the queue in bytes rather than in packets.
- **thresh** - The minimum threshold for the average queuesize.
- **maxthresh** - The maximum threshold for the average queuesize.
- **mean_pktsize** - A rough estimate of the average packet size in bytes. Used in updating the calculated
  average queue size after an idle period.
- **q_weight** - The queue weight, used in the exponential-weighted moving average for calculating the average queue size.
- **wait** - Set to true to maintain an interval between dropped packets.
- **linterm** - As the average queue size varies between "thresh" and "maxthresh", the packet dropping probability varies between 0 and "1/linterm".
- **setbit** - Set to true to set the congestion indication bit in packet headers rather than drop packets.
- **drop-tail** - Set to true to use drop-tail rather than random-drop when the queue overflows.
- **doubleq** - Set to true to give priority to small packets. The default is false.
- **dqthresh** - The largest size in bytes of a "small" packet. This is only used if "doubleq" is set to true.

State Variables

None of the state variables of the RED implementation are accessible.

CBQ Objects

CBQ objects are a subclass of link objects that implement class-based queueing.

$cbq insert $class
Insert traffic class class into the link-sharing structure associated with link object cbq.

$cbq bind $class $classID
Bind class ID classID to the traffic class class associated with link object cbq.

CBQ Configuration Parameters

algorithm - Set to "0" for Ancestor-Only link-sharing, to "1" for Top-Level link-sharing, to "2" for Formal link-sharing. max-pktsize. Used in implementing weighted round-robin.

WRR-CBQ Objects

WRR-CBQ objects are a subclass of CBQ objects that implement weighted round-robin scheduling among classes of the same priority level. In contrast, CBQ objects implement packet-by-packet round-robin scheduling among classes of the same priority level.

Class Object

CLASS objects implement the traffic classes associated with CBQ objects.

$class1 parent $class2
Assign traffic class class2 as the parent class of the traffic class class1. The root class should have parent class "none".

$class1 borrow $class2
Assign traffic class class2 as the class to borrow bandwidth from for the traffic class class1. A class that is not allowed to borrow bandwidth should have borrow class "none".

Class Configuration Parameters

- **priority** - The class's priority level for packet scheduling. Priority-0 classes have the highest priority.
- **depth** - Used for the Top-Level link-sharing algorithm. Leaf classes have "depth=0".
- **allotment** - The link-sharing bandwidth allocated to the class, given as a fraction of the link bandwidth.
- **maxidle** - Used in calculating the bandwidth used by the class.
- **minidle** - Used in calculating the bandwidth used by the class.
- **extradelay** - Used in delaying an overlimit class. For a further explanation of the CBQ variables, see [5] and [6].

TCP OBJECTS

TCP objects are a subclass of agent objects that implement the BSD Tahoe TCP transport protocol. They inherit all of the generic agent functionality.

$tcp source ftp
This creates an ftp source

$tcp source telnet
This creates a telnet source

$tcp source bursty
This creates a bursty TCP source

Install a data source of the type indicated in the tcp agent. Returns the name of the Tcl object that corresponds to the new source.

TCP Configuration Parameters

- **window** - The upper bound on the advertised window for the TCP connection.
- **window-init** - The initial size of the congestion window on slow-start.
- **window-option** - The algorithm to use for managing the congestion window.
- **window-thresh** - Gain constant to exponential averaging filter used to compute awnd (see below). For investigations of different window-increase algorithms.
- **overhead** - The range of a uniform random variable used to delay each output packet. The idea is to insert random delays at the source in order to avoid phase effects, when desired [4]. This has only been implemented for the Tahoe ("tcp") version of tcp, not for tcp-reno. This is not intended to be a realistic model of CPU processing overhead.
- **ecn** - Set to true to use explicit congestion notification in addition to packet drops to signal congestion.
- **packet-size** - The size in bytes to use for all packets from this source.
- **tcp-tick** - The TCP clock granularity for measuring roundtrip times.
- **bug-fix** - Set to true to remove a bug when multiple fast retransmits are allowed for packets dropped in a single window of data.
- **maxburst** - Set to zero to ignore. Otherwise, the maximum number of packets that the source can send in response to a single incoming ACK.
- **MWS** - The Maximum Window Size in packets for a TCP connection. MWS determines the size of an array in tcp-sink.cc. The default for MWS is 1024 packets. For Tahoe TCP, the "window" parameter, representing the receiver's advertised window, should be less than MWS-1. For Reno TCP, the "window" parameter should be less than (MWS-1)/2. (MWS is currently a defined constant, but we plan to change MWS to a configuration parameter in a future release.)

State Variables

- **dupacks** - Number of duplicate acks seen since any new data was acknowledged.
- **seqno** - Current position in the sequence space (can move backwards).
- **ack** - Highest acknowledgment seen from receiver.
- **cwnd** - Current value of the congestion window.
- **awnd** - Current value of a low-pass filtered version of the congestion window. For investigations of different window-increase algorithms.
- **ssthresh** - Current value of the slow-start threshold.
- **rtt** - Round-trip time estimate.
- **srtt** - Smoothed round-trip time estimate.
- **rttvar** - Round-trip time mean deviation estimate.
- **backoff** - Round-trip time exponential backoff constant.

TCP FACK OBJECTS

TCP Fack objects are a subclass of TCP objects that implement Forward Acknowledgment congestion control. They inherit all of the TCP object functionality.

FACK TCP Configuration Parameters

ss-div4 - Overdamping algorithm. Divides ssthresh by 4 (instead of 2) if congestion is detected within 1/2 RTT of slow-start. (1=Enable, 0=Disable)

rampdown - Rampdown data smoothing algorithm. Slowly reduces congestion window rather than instantly halving it. (1=Enable, 0=Disable)

SOURCE OBJECTS

Source objects create data for a transport object to send (e.g., TCP).

$source start
Causes the data source to start producing an unbounded amount of data.

$source produce n
Causes the source to produce exactly n packets instantaneously.

$source stop
Causes the data source to stop

Source Configuration Parameters

- **maxpkts** - The maximum number of packets generated by the source.

TCP-SINK Objects

Tcp-sink objects are a subclass of agent objects that implement a receiver for TCP packets. The simulator only implements "one-way" TCP connections, where the TCP source sends data packets and the TCP sink sends ACK packets. Tcp-sink objects inherit all of the generic agent functionality. There are no methods or state variables specific to the tcp-sink object.

Sink Configuration Parameters

packet-size - The size in bytes to use for all acknowledgment packets.

TCP-SINK-DA Objects

Tcp-sink-da objects are a subclass of tcp-sink that implement a delayed-ACK receiver for TCP packets. They inherit all of the tcp-sink object functionality. There are no methods or state variables specific to the tcp-sink-da object.

Del Sink Configuration Parameters

interval - The amount of time to delay before generating an acknowledgment for a single packet. If another packet arrives before this time expires, generate an acknowledgment immediately.

CONSTANT BIT-RATE Objects

Cbr objects generate packets at a constant bit rate. They inherit all of the generic agent functionality. There are no state variables specific to the cbr class.

$cbr start - Causes the source to start generating packets.

$cbr stop - Causes the source to stop generating packets.

CBR Configuration Parameters

interval - The amount of time to delay between packet transmission times.

packet-size - The size in bytes to use for all packets from this source.

TRACE Objects

Trace objects are used to generate event level capture logs, either directly to an output file, indirectly through a Tcl procedure, or both. There are no state variables or configuration parameters specific to the trace class.

$trace attach fileID

Attach a file to a trace object so that events are written to the indicated file. fileID must be a file handle returned by the Tcl open command and it must have been open for writing.

$trace detach

Detach any attached file indicated from the trace object. Events will no longer be logged to this file.

$trace callback proc

Arrange for the Tcl procedure proc to be called for every event logged by the trace object. Both the Tcl callback and an attached file may be simultaneously active. proc is called with a single argument, which consists of a Tcl list representing the captured event. Be warned that invoking the Tcl interpreter on each event like this will substantially slow down the simulation. If proc is an empty string, cancel the callback.

Trace records for link objects have the following form:

<code> <time> <hsrc> <packet> where

<code> := [hd+-] h=hop d=drop +=enque -=deque

<time> := simulation time in seconds
<hsrc> := first node address of hop/queuing link
<hdst> := second node address of hop/queuing link
<packet> := <type> <size> <flags> <class> <src.sport> <dst.dport>

<type> := tcp | telnet | cbr | ack etc.
<size> := packet size in bytes
<flags> := [CP] C=congestion, P=priority
<class> := class ID number
<src.sport> := transport address (src=node,sport=agent)
<dst.sport> := transport address (dst=node,dport=agent)

For links that use RED gateways, there are additional trace records as follows:

<code> <time> <value> where

<code> := [Qap] Q=queue size, a=average queue size, p=packet dropping probability
<time> := simulation time in seconds
<value> := value

A Step-by-step walk through Example

Example 1

```
#
# Create two nodes and connect them with a 1.5Mb link with a
# transmission delay of 10ms using FIFO drop-tail queuing
#
set n0 [ns node]
set n1 [ns node]
ns_duplex $n0 $n1 1.5Mb 10ms drop-tail

#
# Set up BSD Tahoe TCP connections in opposite directions.
#
set src1 [ns agent tcp $n0]
set snk1 [ns agent tcp-sink $n1]
set src2 [ns agent tcp $n1]
set snk2 [ns agent tcp-sink $n0]
ns_connect $src1 $snk1
ns_connect $src2 $snk2
$src1 set class 1
$src2 set class 2

#
# Create ftp sources at the each node
#
set ftp1 [$src1 source ftp]
set ftp2 [$src2 source ftp]

#
# Start up the first ftp at the time 0 and
# the second ftp staggered 1 second later
#
```

```
ns at 0.0 "$ftp1 start"
ns at 1.0 "$ftp2 start"

#
# Create a trace and arrange for all link
# events to be dumped to "out.tr"
#
set trace [ns trace]
$trace attach [open out.tr w]
foreach link [ns link] {
      $link trace $trace
}

#
# Dump the queuing delay on the n0/n1 link
# to stdout every second of simulation time.
#
proc dump { link interval } {
       ns at [expr [ns now] + $interval] "dump $link $interval"
       set delay [expr 8 * [$link integral qsize] / [$link get bandwidth]]
       puts "[ns now] delay=$delay"
}
ns at 0.0 "dump [ns link $n0 $n1] 1"

#
# run the simulation for 10 simulated seconds
#
ns at 10.0 "exit 0"
ns run
```

Example 2

```
#     ATM Network
#
#    output     shared
#       n1---------n2
#          8Mb,10ms
#
#Create a simple 2 switch network with min-hop routing
#Set buffer size to 16 for n1
#
proc create_test_atmnet { } {
        global r1 n1 n2 s1 s2 k1
        set r1 [ns node atm-network min-hop]
        set n1 [ns node atm-switch output early]
        set n2 [ns node atm-switch shared simple]
        $n1 buffer-size 16
        $r1 add-node $n1
        $r1 add-node $n2
        set L [ns_duplex $n1 $n2 8Mb 10ms atm]

#connect the nodes to the external TCP nodes
        ns connect $s1 $r1 $n1
        ns connect $s2 $r1 $n2
        ns connect $k1 $r1 $n2
}
#
# Create a simple four node topology:
#
#        s1
#         \
```

```
#   8Mb,5ms \  0.8Mb,100ms
#            r1 --------- k1
#   8Mb,5ms /
#          /
#         s2
#
global s1 s2 r1 k1 n1 n2
set s1 [ns node]
set s2 [ns node]
set k1 [ns node]

#Call the above function which will set r1 to be the ATM Network
create_test_atmnet

ns_duplex $s1 $r1 8Mb 5ms drop-tail
ns_duplex $s2 $r1 8Mb 5ms drop-tail

#Since we have to set the queue limit of this link so assign it to a variable
set L [ns_duplex $r1 $k1 800kb 100ms drop-tail]

#L is an array of two links. So seperately set the two queue limits.
[lindex $L 0] set queue-limit 6
[lindex $L 1] set queue-limit 6

#Create a TCP sink and set its variables
set tcp1 [ns_create_connection tcp $s1 tcp-sink $k1 0]
$tcp1 set window 50
set ftp1 [$tcp1 source ftp]
ns at 0.0 "$ftp1 start"

tcpDump $tcp1 1.0

proc openTrace { stopTime testName } {
        exec rm -f out.tr temp.rands
        global r1 k1
        set traceFile [open out.tr w]
        ns at $stopTime \
                "close $traceFile ; finish $testName"
        set T [ns trace]
        $T attach $traceFile
        return $T
}

# trace only the bottleneck link
[ns link $r1 $k1] trace [openTrace 5.0 test_tahoe]

ns run
```

References

- S. Keshav, ‘‘REAL: A Network Simulator’’. UCB CS Tech Report 88/472, December 1988. See http://min-
nie.cs.adfa.oz.au/REAL/index.html for more information.
- Floyd, S. and Jacobson, V. Random Early Detection gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, Vol. 1, No. 4. August 1993. pp. 197-413. Available from

http://www-nrg.ee.lbl.gov/floyd/red.html.
- Floyd, S. Simulator Tests. July 1995. URL ftp://ftp.ee.lbl.gov/papers/simtests.ps.Z.
- Floyd, S., and Jacobson, V. On Traffic Phase Effects in Packet-Switched Gateways. Internetworking: Research and Experience, V.3 N.3, September 1992. pp. 115-156.
- Floyd, S., and Jacobson, V. Link-sharing and Resource Management Models for Packet Networks. IEEE/ACM Transactions on Networking, Vol. 3 No. 4,
- Floyd, S., Notes of Class-Based Queueing: Setting Parameters. URL ftp://ftp.ee.lbl.gov/papers/params.ps.Z. September 1995.
- Fall, K., and Floyd, S. Comparisons of Tahoe, Reno, and Sack TCP. December 1995. URL ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z.
- Mathis, M., and Mahdavi, J. Forward Acknowledgement: Refining TCP Congestion Control. ACM Computer Communications Review, Vol. 26 No. 4, October 1996. pp. 281-291. Available from ftp://ftp.psc.edu/pub/networking/papers/Fack.9608.ps