NAME

ns − network simulator (version 2)

SYNOPSIS

ns [*file* [*arg arg ...*]]

DESCRIPTION

ns is an event-driven network simulator. An extensible simulation engine is implemented in C++ that uses MIT's Object Tool Command Language, OTcl (an object oriented version of Tcl) as the command and configuration interface. A previous version of the simulator i.e. ns version 1 used the Tool Command Language, Tcl as the configuration language. The current version still supports simulation scripts written in Tcl meant for the ns version 1 simulator.

This manual page documents some of the interfaces for ns. For much more complete documentation, please see "ns Notes and Documentation" [13], available in the distribution and on the web.

The simulator is invoked via the *ns* interpreter, an extension of the vanilla *otclsh* command shell. A simulation is defined by a OTcl script. The scripts use the Simulator Class as the principal interface to the simulation engine. Using the methods defined in this class, a network topology is defined, traffic sources and sinks are configured, the simulation is invoked, and the statistics are collected. By building upon a fully functional language, arbitrary actions can be programmed into the configuration.

The first step in the simulation is to acquire an instance of the Simulator class. Instances of objects in classes are created and destroyed in ns using the *new* and *delete* methods. For example, an instance of the Simulator object is created by the following command:

```
e.g. set ns [new Simulator]
```

A network topology is realized using three primitive building blocks: nodes, links, and agents. The Simulator class has methods to create/ configure each of these building blocks. Nodes are created with the *node* Simulator method that automatically assigns an unique address to each node. Links are created between nodes to form a network topology with the *simplex-link* and *duplex-link* methods that set up unidirectional and bidirectional links respectively. Agents are the objects that actively drive the simulation. *Agents* can be thought of as the processes and/or transport entities that run on *nodes* that may be end hosts or routers. Traffic sources and sinks, dynamic routing modules and the various protocol modules are all examples of agents. Agents are created by instantiating objects in the subclass of class Agent i.e., *Agent/type* where type specifies the nature of the agent. For example, a TCP agent is created using the command:

```
set tcp [new Agent/TCP]
```

Once the agents are created, they are attached to nodes with the *attach-agent* Simulator method. Each agent is automatically assigned a port number unique across all agents on a given node (analogous to a tcp or udp port). Some types of agents may have sources attached to them while others may generate their own data. For example, you can attach ‘‘ftp’’ and ‘‘telnet’’ sources to ‘‘tcp’’ agents but ‘‘constant bit-rate’’ agents generate their own data. Applications are attached to agents using the *attach-app* method.

Each object has some configuration parameters associated with it that can be modified. Configuration parameters are instance variables of the object. These parameters are initialized during startup to default values that can simply be read from the instance variables of the object. For example, *$tcp set window_* returns the default window size for the tcp object. The default values for that object can be explicitly overridden by simple assignment either before a simulation begins, or dynamically, while the simulation is in progress. For example the window-size for a particular TCP session can be changed in the following manner.

```
$tcp set window_ 25
```

The default values for the configuration parameters of all the class objects subsequently created can also be changed by simple assignment. For example, we can say

```
Agent/TCP set window_ 30
```

to make all future tcp agent creations default to a window size of 30.

Events are scheduled in ns using the *at* Simulator method that allows OTcl procedures to be invoked at arbitrary points in simulation time. These OTcl callbacks provide a flexible simulation mechanism -- they can be used to start or stop sources, dump statistics, instantiate link failures, reconfigure the network topology etc. The simulation is started via the *run* method and continues until there are no more events to be processed. At this time, the original invocation of the *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 *stop* command or by exiting the script with Tcl's standard *exit* command.

Packets are forwarded along the shortest path route from a source to a destination, where the distance metric is the sum of costs of the links traversed from the source to the destination. The cost of a link is 1 by default; the distance metric is simply the hop count in this case. The cost of a link can be changed with the *cost* Simulator method. A static topology model is used as the default in ns in which the states of nodes/links do not change during the course of a simulation. Network Dynamics could be specified using methods described in NETWORK DYNAMICS METHODS section. Also static unicast routing is the default in which the routes are pre-computed over the entire topology once prior to starting the simulation. Methods to enable and configure dynamic unicast and multicast routing are described in the UNICAST ROUTING METHODS and MULTICAST ROUTING METHODS sections respectively.

NS COMMANDS

This section describes the basic commands to create the building blocks of the simulation (i.e. the node, link and agent objects) and to run the simulation.

The first step in running a simulation as stated before is to acquire an instance of the Simulator class that has methods to configure and run the simulation. Throughout this section the object variable name $ns is used to imply a Simulator object.

$ns node
Create a new node object and return a handle to it.

$ns all-nodes-list
Returns a list of all the node objects defined in the simulation.

$ns simplex-link *node1 node2 bw delay type*
Create a new unidirectional link between *node1* and *node2* with bandwidth *bw* in bits per second and link propagation delay *delay* in seconds. *node1* and *node2* must have already been created with the *node* method. *bw* and *delay* default to 1.5 Mbits/sec and 100 ms respectively. The defaults can be changed by modifying the relevant configuration parameters of the DelayLink Object (see DELAYLINK OBJECTS section). *node1* and *node2* must have already been created with the *node* method. The queuing discipline of the link is specified by *type*, which may be **DropTail, FQ, SFQ, DRR, RED, CBQ,** or **CBQ/WRR.** A DropTail link is a simple FIFO queue which drops the last packet in the queue when the queue overflows. A FQ link is for Fair Queuing (for details see [?]). A SFQ link is for Stochastic Fair Queuing (for details see [?]). A DRR link is for deficit round robin scheduling (for details see [9]). A RED link is a random-early drop queue (for details see [2]). A CBQ link is for class-based queuing using a packet-by-packet round-robin scheduler (for details see [3]). A CBQ/WRR link is for class-based queuing with a weighted round robin scheduler. If multicast routing is used links with interface labels are required. Such links are created by setting Simulator NumberInterfaces_ variable to 1. All the subsequently created links will have interface labels. To disable creation of interfaces simply reset NumberInterfaces_ to 0 (this is the default).

$ns duplex-link *node1 node2 bw delay type*
Create a new bidirectional link between *node1* and *node2* with bandwidth *bw* in bits per second and link propagation delay *delay* in seconds. *node1* and *node2* must have already been created with the *node* method. *bw* and *delay* default to 1.5 Mbits/sec and 100 ms respectively. The defaults can be changed by modifying the relevant configuration parameters of the DelayLink Object (see DELAYLINK OBJECTS section). The queuing discipline of the link is specified by *type,* which may be **DropTail, FQ SFQ, DRR, RED, CBQ,** or **CBQ/WRR.** A DropTail link is a simple FIFO queue which drops the last packet in the queue when the queue overflows. A FQ link is for Fair Queuing (for details see [?]). A SFQ link is for Stochastic Fair Queuing (for details see [?]). A DRR link is for deficit round robin scheduling (for details see [9]). A RED link is a random-early drop queue (for details see [2]). A CBQ link is for class-based queuing using a packet-by-packet round-robin scheduler (for details see [3]). A CBQ/WRR link is for class-based queuing with a weighted round robin scheduler. If multicast routing is used links with interface labels are required. Such links are created by setting Simulator NumberInterfaces_ variable to 1. All the subsequently created links will have interface labels. To disable creation of interfaces simply reset NumberInterfaces_ to 0 (this is the default).

$ns link *node1 node2*
Returns a reference to the link connecting nodes *node1* and *node2.* This is useful for setting link configuration parameters and to invoke tracing methods (see LINK OBJECTS section).

$ns queue-limit *node1 node2 queue-limit*
Set the maximum number of packets that can be queued on the link in the direction from *node1* to *node2* to *queue-limit.* The link between node1 and node2 should have already been created.

$ns delay *node1 node2 time-interval*
Set the latency of the link in the direction from *node1* to *node2* to *time-interval* seconds. The link between node1 and node2 should have already been created.

$ns cost *node1 node2 cost-val*
Assign the cost *cost-val* to the link between nodes *node1* and *node2.* The costs assigned to links are used in unicast route computations. All the links default to a cost of 1.

$ns multi-link *node-list bw delay type*
Connects the nodes specified in *node-list* by a mesh of duplex links (to simulate a broadcast LAN) with bandwidth *bw* in bits per second and link propagation delay *delay* in seconds. *node-list* is a list of node object handles that have already been created with the *node* method. *bw* and *delay* default to 1.5 Mbits/sec and 100 ms respectively. The defaults can be changed by modifying the relevant configuration parameters of the DelayLink Object (see DELAYLINK OBJECTS section). The queuing discipline of the link is specified by *type,* which may be **DropTail, FQ SFQ, DRR, RED, CBQ,** or **CBQ/WRR.** A DropTail link is a simple FIFO queue which drops the last packet in the queue when the queue overflows. A FQ link is for Fair Queuing (for details see [?]). A SFQ link is for Stochastic Fair Queuing (for details see [?]). A DRR link is for deficit round robin scheduling (for details see [9]). A RED link is a random-early drop queue (for details see [2]). A CBQ link is for class-based queuing using a packet-by-packet round-robin scheduler (for details see [3]). A CBQ/WRR link is for class-based queuing with a weighted round robin scheduler.

$ns multi-link-of-interfaces *node-list bw delay type*
Connects the nodes specified in *node-list* by a mesh of duplex links with interfaces (to simulate a broadcast LAN) with bandwidth *bw* in bits per second and link propagation delay *delay* in seconds. *node-list* is a list of node object handles that have already been created with the *node* method. *bw* and *delay* default to 1.5 Mbits/sec and 100 ms respectively. The defaults can be changed by modifying the relevant configuration parameters of the DelayLink Object (see DELAYLINK OBJECTS section). The queuing discipline of the link is specified by *type,* which may be **DropTail, FQ SFQ, DRR, RED, CBQ,** or **CBQ/WRR.** A DropTail link is a simple FIFO queue which drops the last packet in the queue when the queue overflows. A FQ link is for Fair Queuing (for details see [?]). A SFQ link is for Stochastic Fair Queuing (for details see [?]). A DRR link is for deficit round robin scheduling (for details see [9]). A RED link is a random-early

drop queue (for details see [2]). A CBQ link is for class-based queuing using a packet-by-packet round-robin scheduler (for details see [3]). A CBQ/WRR link is for class-based queuing with a weighted round robin scheduler.

new Agent/*type*

Create an Agent of type *type* which may be:

Null	- Traffic Sink
LossMonitor	- Traffic Sink that monitors loss parameters
TCP	- BSD Tahoe TCP
TCP/FullTcp	- Full Reno TCP with two-way connections [11]
TCP/Reno	- BSD Reno TCP
TCP/Newreno	- a modified version of BSD Reno TCP
TCP/Vegas	- Vegas TCP (from U. Arizonia via USC)
TCP/Sack1	- BSD Reno TCP with selective ACKs
TCP/Fack	- BSD Reno TCP with forward ACKs
TCPSink	- standard TCP sink
TCPSink/DelAck	- TCP sink that generates delayed ACKs
TCPSink/Sack1	- TCP sink that generates selective ACKs
TCPSink/Sack1/DelAck	- delayed-ack TCP sink with selective ACKs
UDP	- UDP Transport
RTP	- RTP agent
Session/RTP	-
RTCP	- RTCP agent
IVS/Source	-
IVS/Receiver	-
SRM	-

The methods, configuration parameters and the relevant state variables associated with these objects are discussed in detail in later sections. Note that some agents e.g. TCP or SRM do not generate their own data. Such agents need sources attached to them to generate data (see attach-source and attach-traffic methods in AGENT OBJECTS section).

$ns attach-agent *node agent*

Attach the agent object *agent* to *node.* The *agent* and *node* objects should have already been created.

$ns detach-agent *node agent*

Detach the agent object *agent* from *node.*

$ns connect *src dst*

Establish a two-way connection between the agent *src* and the agent *dst.* Returns the handle to *src* agent. A helper method has been defined to facilitate creating and attaching an agent to each of two nodes and establishing a two-way connection between them. (see BUILTINS section).

$ns use-scheduler *type*

Use an event scheduler of type *type* in the simulations. *type* is one of List, Heap, Calendar, RealTime. The List scheduler is the default. A Heap scheduler uses a heap for event queueing. A Calendar scheduler uses a calendar queue to keep track of events. RealTime scheduler is used in emulation mode when the simulator interacts with an external agent.

$ns at *time procedure*

Evaluate *procedure* at simulation time *time.* The procedure could be a globally accessible function (proc) or an object method (instproc). This command can be used to start and stop sources, dynamically reconfigure the simulator, dump statistics at specified intervals, etc. Returns an event id.

$ns cancel *eid*

Remove the event specified by the event id *eid* from the event queue.

$ns now
: Return the current simulation time.

$ns gen-map
: Walks through the simulation topology and lists all the objects that have been created and the way they are hooked up to each other. This is useful to debug simulation scripts.

ns-version
: Return a string identifying the version of ns currently running. This method is executed in the global context by the interpreter.

ns-random *[seed]*
: If *seed* is not present, return a pseudo-random integer between 0 and 2ˆ31-1. Otherwise, seed the pseudo-random number generator with *seed* and return the seed used. If *seed* is 0, choose an initial seed heuristically (which varies on successive invocations). This method is executed in the global context by the interpreter.

Ns has other facilities for random number generation; please see documentation for details [13].

OBJECT HIERARCHY

A brief description of the object hierarchy in *ns* is presented in this section. This description is not intended to be complete. It has been provided to depict how the methods and configuration parameters associated with the various objects are inherited. For more complete information see "ns notes & documentation" and the automatically generated class library information on the ns web page.

Objects are associated with configuration parameters that can be dynamically set and queried, and state variables that can be queried (usually modified only when the state variables need to be reset for another simulation run).

Configuration parameters represent simulation parameters that are usually fixed during the entire simulation (like a link bandwidth), but can be changed dynamically if desired. State variables represent values that are specific to a given object and that object's implementation.

The following diagram depicts a portion the object hierarchy:

```
Simulator
    MultiSim
Node
Link
    SimpleLink
        CBQLink
    DummyLink
DelayLink
Queue
    DropTail
    FQ
    SFQ
    DRR
    RED
    CBQ
    CBQ/WRR
QueueMonitor
    ED
        Flowmon
        Flow
rtObject
RouteLogic
Agent
    rtProto
```

```
            Static
            Session
            DV
            Direct
        Null
        LossMonitor
        TCP
            FullTcp
            Reno
            Newreno
            Sack1
            Fack
        TCPSink
            DelAck
            Sack1
                DelAck
        UDP
        RTP
        RTCP
        IVS
            Source
            Receiver
        SRM
        Session
            RTP [how is this diff from Agent/CBR/RTP]
    Appplication
        FTP
        Telnet
        Traffic
            Expoo
            Pareto
            CBR
            Trace
    Integrator
    Samples
```

For a complete, automatically generated, object hierarchy, see the link "class hierarchy" (which points to http://www-sop.inria.fr/rodeo/personnel/Antoine.Clerget/ns/) on the ns web pages. (Thanks to Antoine Clerget for maintaining this!)

For example, any method that is supported by a *TCP* agent is also supported by a *Reno* or a *Sack1* agent. Default configuration parameters are also inherited. For example, *$tcp set window_ 20* where $tcp is a TCP agent defines the default TCP window size for both *TCP* and *Reno* objects.

OBJECT METHODS

The following sections document the methods, configuration parameters and state variables associated with the various objects as well as those to enable Network dynamics, Unicast routing, Multicast routing and Trace and Monitoring support. 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.

NODE OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.]

$node id
Returns the node id.

$node neighbors
Returns a list of the neighbour node objects.

$node attach *agent*
Attach an agent of type *agent* to this node.

$node detach *agent*
Detach an agent of type *agent* from this node.

$node agent *port*
Return a handle to the agent attached to port *port* on this node. Returns an empty string if the port is not in use.

$node reset
Reset all agents attached to this node. This would re-initialize the state variables associated with the various agents at this node.

$node rtObject?
Returns a handle to rtObject if there exists an instance of the object at that node. Only nodes that take part in a dynamic unicast routing protocol will have this object (see UNICAST ROUTING METHODS and RTOBJECT OBJECTS section).

$node join-group *agent group*
Add the agent specified by the object handle *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. Multicast group address should be in the range 0x8000 - 0xFFFF.

$node allocaddr
Returns multicast group address in ascending order on each invocation starting from 0x8000 and ending at 0xFFFF.

$node shape *shape*
Set the shape of the node to "*shape*". When called before the simulator starts to run, it changes the default shape of the node in the nam trace file. The default shape of a node is """circle"""

$node color *color*
Set the color of the node to *color*. It can be called anytime to change the current color of the node in nam trace file, if there is one.

$node get-attribute *name*
Get the specified attribute *name* of the node. Currently a Node object has two attributes: *COLOR* and *SHAPE*. Note: these letters must be capital.

$node add-mark *name color shape*
Add a mark (in nam trace file) with *color* and *shape* around the node. The shape can be """circle""", """hexagon""" and """square""" (case sensitive). The added mark will be identified by *name*.

$node delete-mark *name*
Delete the mark with *name* in the given node.

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

LINK OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.]

$link trace-dynamics *ns fileID*
Trace the dynamics of this link and write the output to *fileID* filehandle. *ns* is an instance of the Simulator or MultiSim object that was created to invoke the simulation (see TRACE AND

MONITORING METHODS section for the output trace format).

$link trace-callback *ns cmd*
Trace all packets on the link with the callback *cmd*. Cmd is invoked for each trace event (enqueue, dequeue, drop) with the text that would be logged as parameters. (See the description of the log file for this information.) A demo of trace callbacks is in the program tcl/ex/callback_demo.tcl in the distribution.

$link color *color*
Set the color of the Link object. It can be called anytime to change the current color of the link in nam trace file, if there is one.

$link get-attribute *name*
Get the specified attribute *name* of the Link. Currently a Link object has three attributes: *COLOR*, *ORIENTATION*, and *QUEUE_POS*.

Currently the following two functions should not be directly called. Use **$ns duplex-link-op** instead. Refer to the corresponding section in this man page.

$link orient *ori*
Set the orientation of the link to *ori*. When called before the simulator starts to run, it changes the default orientation of the link in nam trace file, if there is one. If orientation is unspecified for any link(s), nam will use automatic layout. The default orientation of a Link object is unspecified.

$link queuePos *pos*
Set the queue position of the link to *pos*. When called before the simulator starts to run, it changes the default queue placement of the simplex link in nam trace file, if there is one. *pos* specifies the angle between the horizontal line and the line along which queued packets will be displayed.

SIMPLELINK OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.]

$link cost *cost-val*
Make *cost-val* the cost of this link.

$link cost?
Return the cost of this link.

Any configuration parameters or state variables?

DELAYLINK OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.] The DelayLink Objects determine the amount of time required for a packet to traverse a link. This is defined to be size/bw + delay where size is the packet size, bw is the link bandwidth and delay is the link propagation delay. There are no methods or state variables associated with this object.

Configuration Parameters

*bandwidth_*
Link bandwidth in bits per second.

*delay_* Link propagation delay in seconds.

There are no state variables associated with this object.

NETWORK DYNAMICS METHODS

This section describes methods to make the links and nodes in the topology go up and down according to various distributions. A dynamic routing protocol should generally be used whenever a simulation is to be done with network dynamics. Note that a static topology model is the default in ns.

$ns rtmodel *model model-params node1 [node2]*

Make the link between *node1* and *node2* change between up and down states according to the model *model.* In case only *node1* is specified all the links incident on the node would be brought up and down according to the specified *model. model-params* contains the parameters required for the relevant model and is to be specified as a list i.e. the parameters are to be enclosed in curly brackets. *model* can be one of *Deterministic, Exponential, Manual, Trace.* Returns a handle to a model object corresponding to the specified *model.*

In the Deterministic model *model-params* is *[start-time] up-interval down-interval [finish-time].* Starting from *start-time* the link is made up for *up-interval* and down for *down-interval* till *finish-time* is reached. The default values for start-time, up-interval, down-interval are 0.5s, 2.0s, 1.0s respectively. finish-time defaults to the end of the simulation. The start-time defaults to 0.5s in order to let the routing protocol computation quiesce.

If the Exponential model is used *model-params* is of the form *up-interval down-interval* where the link up-time is an exponential distribution around the mean *up-interval* and the link down-time is an exponential distribution around the mean *down-interval.* Default values for *up-interval* and *down-interval* are 10s and 1s respectively.

If the Manual distribution is used *model-params* is *at op* where *at* specifies the time at which the operation *op* should occur. *op* is one of *up, down.* The Manual distribution could be specified alternately using the *rtmodel-at* method described later in the section.

If Trace is specified as the *model* the link/node dynamics is read from a Tracefile. The *model-params* argument would in this case be the file-handle of the Tracefile that has the dynamics information. The tracefile format is identical to the trace output generated by the trace-dynamics link method (see TRACE AND MONITORING METHODS SECTION).

$ns rtmodel-delete *model-handle*

Delete the instance of the route model specified by *model-handle.*

$ns rtmodel-at *at op node1 [node2]*

Used to specify the up and down times of the link between nodes *node1* and *node2.* If only *node1* is given all the links incident on *node1* will be brought up and down. *at* is the time at which the operation *op* that can be either *up* or *down* is to be performed on the specified link(s).

QUEUE OBJECTS

A queue object is a general class of object capable of holding and possibly marking or discarding packets as they travel through the simulated topology.

Configuration Parameters

*limit_* The queue size in packets.

*blocked_*
Set to false by default, this is true if the queue is blocked (unable to send a packet to its downstream neighbor).

*unblock_on_resume_*
Set to true by default, indicates a queue should unblock itself at the time the last packet packet sent has been transmitted (but not necessarily received).

DROP-TAIL OBJECTS

Drop-tail objects are a subclass of Queue objects that implement simple FIFO queue. There are no methods that are specific to drop-tail objects. The only configuration parameter is *drop-front_*, which when set to true causes the queue to behave as a drop-from-front queueing discipline. This variable is set to false by default.

FQ OBJECTS

FQ objects are a subclass of Queue objects that implement Fair queuing. There are no methods that are specific to FQ objects.

Configuration Parameters

*secsPerByte_*

There are no state variables associated with this object.

SFQ OBJECTS

SFQ objects are a subclass of Queue objects that implement Stochastic Fair queuing. There are no methods that are specific to SFQ objects.

Configuration Parameters

*maxqueue_*

*buckets_*

There are no state variables associated with this object.

DRR OBJECTS

DRR objects are a subclass of Queue objects that implement deficit round robin scheduling. These objects implement deficit round robin scheduling amongst different flows (A particular flow is one which has packets with the same node and port id OR packets which have the same node id alone). Also unlike other multi-queue objects, this queue object implements a single shared buffer space for its different flows.

Configuration Parameters

*buckets_*
Indicates the total number of buckets to be used for hashing each of the flows.

*blimit_* Indicates the shared buffer size in bytes.

*quantum_*
Indicates (in bytes) how much each flow can send during its turn.

*mask_* mask_, when set to 1, means that a particular flow consists of packets having the same node id (and possibly different port ids), otherwise a flow consists of packets having the same node and port ids.

RED OBJECTS

RED objects are a subclass of Queue objects that implement random early-detection gateways. The object can be configured to either drop or ‘‘mark’’ packets. There are no methods that are specific to RED objects.

Configuration Parameters

*bytes_* Set to "true" to enable ‘‘byte-mode’’ RED, where the size of arriving packets affect the likelihood of marking (dropping) packets.

*queue-in-bytes_*
Set to "true" to measure the average queue size in bytes rather than packets. Enabling this option also causes *thresh_* and *maxthresh_* to be automatically scaled by *mean_pkt-size_* (see below).

*thresh_* The minimum threshold for the average queue size in packets.

*maxthresh_*
The maximum threshold for the average queue size in packets.

*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 mark packets by setting the congestion indication bit in packet headers rather than drop packets.

*drop-tail_*
Set to true to use drop-tail rather than random-drop or drop-from-front when the queue overflows or the average queue size exceeds "maxthresh_". This is the default behavior. For a further explanation of these variables, see [2].

*drop-rand_*
Set to true to use random-drop rather than drop-tail or drop-from-front when the queue overflows or the average queue size exceeds "maxthresh_".

*drop-front_*
Set to true to use drop-from-front rather than drop-tail or random drop when the queue overflows or the average queue size exceeds "maxthresh_".

*ns1-compat_*
Set to true to avoid resetting the count since the last packet drop, after a forced packet is dropped. This gives compatibility with previous behavior of RED. The default is set to false.

entle_ Set to true to increase the packet drop rate slowly from max_p to 1 as the average queue size ranges from maxthresh to twice maxthresh. The default is set to false, and max_p increases abruptly from max_p to 1 when the average queue size exceeds maxthresh.

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

CBQ OBJECTS

CBQ objects are a subclass of Queue 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 $cbqclass $id1 [$id2]
Cause packets containing flow id *$id1* (or those in the range *$id1* to *$id2* inclusive) to be associated with the traffic class *$cbqclass.*

$cbq algorithm $alg
Select the CBQ internal algorithm. *$alg* may be set to one of: "ancestor-only", "top-level", or "formal".

CBQ/WRR OBJECTS

CBQ/WRR 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.

Configuration Parameters

*maxpkt_*
The maximum size of a packet in bytes. This is used only by CBQ/WRR objects in computing maximum bandwidth allocations for the weighted round-robin scheduler.

CBQCLASS OBJECTS

CBQClass objects implement the traffic classes associated with CBQ objects.

$cbqclass setparams *parent okborrow allot maxidle prio level extradelay*
Sets several of the configuration parameters for the CBQ traffic class (see below).

$cbqclass parent [$cbqcl|none]
specify the parent of this class in the link-sharing tree. The parent may be specified as ‘‘none’’ to indicate this class is a root.

$cbqclass newallot $a
Change the link allocation of this class to the specified amount (in range 0.0 to 1.0). Note that only the specified class is affected.

$cbqclass install-queue $q
Install a Queue object into the compound CBQ or CBQ/WRR link structure. When a CBQ object is initially created, it includes no internal queue (only a packet classifier and scheduler).

Configuration Parameters

okborrow_
is a boolean indicating the class is permitted to borrow bandwidth from its parent.

allot_ is the maximum fraction of link bandwidth allocated to the class expressed as a real number between 0.0 and 1.0.

maxidle_
is the maximum amount of time a class may be required to have its packets queued before they are permitted to be forwarded

priority_
is the class’ priority level with respect to other classes. This value may range from 0 to 10, and more than one class may exist at the same priority. Priority 0 is the highest priority.

level_ is the level of this class in the link-sharing tree. Leaf nodes in the tree are considered to be at level 1; their parents are at level 2, etc.

extradelay_
increase the delay experienced by a delayed class by the specified number of seconds.

QUEUEMONITOR Objects

QueueMonitor Objects are used to monitor a set of packet and byte arrival, departure and drop counters. It also includes support for aggregate statistics such as average queue size, etc. [see TRACE AND MONITORING METHODS].

$queuemonitor reset
reset all the cumulative counters described below (arrivals, departures, and drops) to zero. Also, reset the integrators and delay sampler, if defined.

$queuemonitor set-delay-samples *delaySamp_*
Set up the Samples object *delaySamp_* to record statistics about queue delays. *delaySamp_* is a handle to a Samples object i.e the Samples object should have already been created.

$queuemonitor get-bytes-integrator
Returns an Integrator object that can be used to find the integral of the queue size in bytes. (see Integrator Objects section).

$queuemonitor get-pkts-integrator
Returns an Integrator object that can be used to find the integral of the queue size in packets. (see Integrator Objects section).

$queuemonitor get-delay-samples
Returns a Samples object *delaySamp_* to record statistics about queue delays (see Samples Objects section).

There are no configuration parameters specific to this object.

State Variables

*size_* Instantaneous queue size in bytes.

*pkts_* Instantaneous queue size in packets.

*parrivals_*
Running total of packets that have arrived.

*barrivals_*
Running total of bytes contained in packets that have arrived.

*pdepartures_*
Running total of packets that have departed (not dropped).

*bdepartures_*
Running total of bytes contained in packets that have departed (not dropped).

*pdrops_*
Total number of packets dropped.

*bdrops_*
Total number of bytes dropped.

*bytesInt_*
Integrator object that computes the integral of the queue size in bytes. The *sum_* variable of this object has the running sum (integral) of the queue size in bytes.

*pktsInt_*
Integrator object that computes the integral of the queue size in packets. The *sum_* variable of this object has the running sum (integral) of the queue size in packets.

QUEUEMONITOR/ED Objects

This derived object is capable of differentiating regular packet drops from *early* drops. Some queues distinguish regular drops (e.g. drops due to buffer exhaustion) from other drops (e.g. random drops in RED queues). Under some circumstances, it is useful to distinguish these two types of drops.

State Variables

*epdrops_*
The number of packets that have been dropped ‘‘early’’.

*ebdrops_*
The number of bytes comprising packets that have been dropped ‘‘early’’

Note: because this class is a subclass of QueueMonitor, objects of this type also have fields such as `pdrops_` and `bdrops_`. These fields describe the *total* number of dropped packets and bytes, including both early and non-early drops.

QUEUEMONITOR/ED/FLOWMON Objects

These objects may be used in the place of a conventional QueueMonitor object when wishing to collect per-flow counts and statistics in addition to the aggregate counts and statistics provided by the basic QueueMonitor.

$fmon classifier [$cl]
insert (read) the specified classifier into (from) the flow monitor object. This is used to map incoming packets to which flows they are associated with.

$fmon dump
Dump the current per-flow counters and statistics to the I/O channel specified in a previous `attach` operation.

$fmon flows
Return a character string containing the names of all flow objects known by this flow monitor. Each of these objects are of type QueueMonitor/ED/Flow.

$fmon attach $chan
Attach a tcl I/O channel to the flow monitor. Flow statistics are written to the channel when the `dump` operation is executed.

Configuration Parameters

enable_in_
Set to true by default, indicates that per-flow arrival state should be kept by the flow monitor. If set to false, only the aggregate arrival information is kept.

enable_out_
Set to true by default, indicates that per-flow departure state should be kept by the flow monitor. If set to false, only the aggregate departure information is kept.

enable_drop_
Set to true by default, indicates that per-flow drop state should be kept by the flow monitor. If set to false, only the aggregate drop information is kept.

enable_edrop_
Set to true by default, indicates that per-flow early drop state should be kept by the flow monitor. If set to false, only the aggregate early drop information is kept.

QUEUEMONITOR/ED/FLOW Objects

These objects contain per-flow counts and statistics managed by a QUEUEMONITOR/ED/FLOWMON object. They are generally created in an OTcl callback procedure when a flow monitor is given a packet it cannot map on to a known flow. Note that the flow monitor's classifier is responsible for mapping packets to flows in some arbitrary way. Thus, depending on the type of classifier used, not all of the state variables may be relevant (e.g. one may classify packets based only on flow id, in which case the source and destination addresses may not be significant).

State Variables

src_ The source address of packets belonging to this flow.

dst_ The destination address of packets belonging to this flow.

flowid_
The flow id of packets belonging to this flow.

UNICAST ROUTING METHODS

A dynamic unicast routing protocol can be specified to run on a subset of nodes in the topology. Note that a dynamic routing protocol should be generally used whenever a simulation is done with network dynamics.

$ns rtproto *proto node-list*
Specifies the dynamic unicast routing protocol *proto* to be run on the nodes specified by *node-list*. Currently *proto* can be one of Static, Session, DV. Static routing is the default. Session implies that the unicast routes over the entire topology are instantaneously recomputed whenever a link goes up or down. DV implies that a simple distance vector routing protocol is to be simulated.

node-list defaults to all the nodes in the topology.

$ns compute-routes

Compute routes between all the nodes in the topology. This can be used if static routing is done and the routes have to be recomputed as the state of a link has changed. Note that Session routing (see *rtproto* method above) will recompute routes automatically whenever the state of any link in the topology changes.

$ns get-routelogic

Returns an handle to a RouteLogic object that has methods for route table lookup etc.

ROUTELOGIC OBJECTS

$routelogic lookup *srcid destid*

Returns the id of the node that is the next hop from the node with id *srcid* to the node with id *destid.*

$routelogic dump *nodeid*

Dump the routing tables of all nodes whose id is less than *nodeid.* Node ids are typically assigned to nodes in ascending fashion starting from 0 by their order of creation.

RTOBJECT OBJECTS

Every node that takes part in a dynamic unicast routing protocol will have an instance of rtObject (see NODE OBJECTS section for the method to get an handle to this object at a particular node). Note that nodes will not have an instance of this object if Session routing is done as a detailed routing protocol is not being simulated in this case.

$rtobject dump-routes *fileID*

Dump the routing table to the output channel specified by *fileID. fileID* must be a file handle returned by the Tcl *open* command and it must have been opened for writing.

$rtobject rtProto? *proto*

Returns a handle to the routing protocol agent specified by *proto* if it exists at that node. Returns an empty string otherwise.

$rtobject nextHop? *destID*

Returns the id of the node that is the next hop to the destination specified by the node id, *destID.*

$rtobject rtpref? *destID*

$rtobject metric? *destID*

MULTICAST ROUTING METHODS

Multicast routing is enabled by setting Simulator EnableMcast_ variable to 1 at the beginning of the simulation. Note that this variable must be set before any node, link or agent objects are created in the simulation. Also links must have been created with interface labels (see simplex-link and duplex-link methods in NS COMMANDS section).

$ns mrtproto *proto node-list*

Specifies the multicast routing protocol *proto* to be run on the nodes specified by *node-list.* Currently *proto* can be one of CtrMcast, DM, detailedDM, dynamicDM, pimDM. *node-list* defaults to all the nodes in the topology. Returns an handle to a protocol-specific object that has methods, configuration parameters specific to that protocol. Note that currently CtrMcastComp object is returned if CtrMcast is used but a null string is returned if DM, detailedDM, dynamicDM or pimDM are used.

If proto is ’CtrMcast’ a Rendezvous Point (RP) rooted shared tree is built for a multicast group. The actual sending of prune, join messages etc. to set up state at the nodes is not simulated. A centralized computation agent is used to compute the fowarding trees and set up multicast forwarding state, (*,G) at the relevant nodes as new receivers join a group. Data packets from the senders to a group are unicast to the RP. Methods are provided in the CtrMcastComp object (see CTRMCASTCOMP OBJECTS section) that is returned by mrtproto to switch to source-specific trees, choose some nodes as candidate RPs etc. When a node/link on a multicast distribution tree goes down, the tree is instanteously recomputed.

If proto is ’DM’ DVMRP-like dense mode is simulated. Parent-child lists are used to reduce the number of links over which the data packets are broadcast. Prune messages are sent by nodes to remove branches from the multicast forwarding tree that do not lead to any group members. The prune timeout value is 0.5s by default (see DM OBJECTS section to change the default). This does not adapt to network changes. There is also currently no support for proper functioning in topologies with LANs.

If proto is ’detailedDM’ a dense mode protocol based on Protocol Independent Multicast - Dense Mode (PIM-DM) is simulated. This is currently the most complete version of the dense mode protocol in the simulator and is recommended for use over the other dense mode protocols. It adapts to network dynamics and functions correctly in topologies with LANs (where LANs are created using the multi-link-of-interfaces method - see NS COMMANDS). In case there are multiple potential forwarders for a LAN, the node with the highest id is chosen as the forwarder (this is done through the Assert mechanism). The default values for the prune timeout, interface deletion timeout (used for LANs) and graft retransmission timeout are 0.5s, 0.1s and 0.05s respectively. (see Prune/Iface/Timer, Deletion/Iface/Timer and GraftRtx/Timer objects respectively to change the default values and for more information about the timers).

If proto is ’dynamicDM’ DVMRP-like dense mode protocol that adapts to network changes is simulated. (i.e. the information that a particular neighbouring node uses this node to reach a particular network) is read from the routing tables of neighbouring nodes in order to adapt to network dynamics (DVMRP runs its own unicast routing protocol that exchanges this information). The current implementation does not support proper functioning in topologies with LANs. The prune timeout value is 0.5s by default (see DM OBJECTS section to change the default).

If proto is ’pimDM’ Protocol Independent Multicast - Dense mode is simulated. In this case the data packets are broadcast over all the outgoing links except the incoming link. Prune messages are sent by nodes to remove the branches of the multicast forwarding tree that do not lead to any group members. The current implementation does not adapt to network dynamics and does not support proper functioning in topologies with LANs. The prune timeout value is 0.5s by default (see DM OBJECTS section to change the default).

CTRMCASTCOMP OBJECTS

A handle to the CtrMcastComp object is returned when the protocol is specified as ’CtrMcast’ in mrtproto.

$ctrmcastcomp switch-treetype *group-addr*

Switch from the Rendezvous Point rooted shared tree to source-specific trees for the group specified by *group-addr.* Note that this method cannot be used to switch from source-specific trees back to a shared tree for a multicast group.

$ctrmcastcomp set_c_rp *node-list*

Make all the nodes specified in *node-list* as candidate RPs and change the state of all the other nodes to not be candidate RPs. Note that all nodes are candidate RPs by default. Currently the node with the highest node id serves as the RP for all multicast groups. This method should be invoked before any source starts sending packets to the group or any receiver joins the group.

$ctrmcastcomp get_rp *node group*

Returns the RP for the group as seen by the node *node* for the multicast group with address *group-addr.* Note that different nodes may see different RPs for the group if the network is partitioned as the nodes might be in different partitions.

DM OBJECTS

DM Objects implement DVMRP style densemode multicast where parent-child lists are used to reduce the number of links over which initial data packets are broadcast. There are no methods or state variables specific to this object.

Configuration parameters

PruneTimeout

Timeout value for the prune state at nodes.

PRUNE/IFACE/TIMER OBJECTS

The Prune/Iface/Timer objects are used to implement the prune timer for detailedDM. There are no methods or state variables specific to this object.

Configuration parameters

timeout

Timeout value for the prune state at nodes.

DELETION/IFACE/TIMER OBJECTS

The Deletion/Iface/Timer objects are used to implement the interface deletion timer that are required for correct functioning at nodes that are part of LANs. If a node has a LAN as its incoming interface for packets from a certain source and it does not have any downstream members it sends out a prune message onto the LAN. Any node that has the LAN as its incoming interface for the same source and has downstream members on hearing the prune message sent on the LAN. will send a join message onto the LAN. When the node that is acting as the forwarder for the LAN hears the prune message from the LAN, it does not immediately prune off the LAN as its outgoing interface. Instead it starts an interface deletion timer for the outgoing interface. The forwarder will remove the LAN as its outgoing interface only if it does not receive any join messages from the LAN before its deletion timer expires. There are no methods or state variables specific to this object.

Configuration parameters

timeout

Timeout value for the interface deletion timer.

GRAFTRTX/TIMER OBJECTS

The GraftRtx/Timer objects are used to implement the graft retransmission timer at nodes. This is to ensure the reliability of grafts sent upstream by a node.

Configuration parameters

timeout

Timeout value for the graft retransmission timer.

AGENT OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.]

$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 destination node this agent is connected to.

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

$agent attach-source *type*

Install a data source of type *type* in this agent. *type* is one of FTP or bursty[???]. See the corresponding object methods for information on configuration parameters. Returns a handle to the source object.

$agent attach-traffic *traffic-object*

Attach *traffic-object* to this agent *traffic-object* is an instance of Traffic/Expoo, Traffic/Pareto or Traffic/Trace. Traffic/Expoo generates traffic based on an Exponential On/Off distribution. Traffic/Pareto generates traffic based on a Pareto On/Off distribution. Traffic/Trace generates traffic from a trace file. The relevant configuration parameters for each of the above objects can be found in the TRAFFIC METHODS section.

$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.

Configuration Parameters

*dst_*	Address of destination that the agent is connected to. Currently 32 bits with the higher 24 bits the destination node ID and the lower 8 bits being the port number.

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

NULL OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.] Null objects are a subclass of agent objects that implement a traffic sink. They inherit all of the generic agent object functionality. There are no methods, configuration parameters or state variables specific to this object.

LOSSMONITOR OBJECTS

[NOTE: This section has not been verified to be up-to-date with the release.] LossMonitor objects are a subclass of agent objects that implement a traffic sink which also maintains some statistics about the received data e.g., number of bytes received, number of packets lost etc. They inherit all of the generic agent object functionality.

$lossmonitor clear

Resets the expected sequence number to -1.

Configuration Parameters

There are no configuration parameters specific to this object.

State Variables

*nlost_*	Number of packets lost.
*npkts_*	Number of packets received.
*bytes_*	Number of bytes received.
*lastPktTime_*	Time at which the last packet was received.
*expected_*	The expected sequence number of the next packet.

TCP OBJECTS

TCP objects are a subclass of agent objects that implement the BSD Tahoe TCP transport protocol as described in [7]. They inherit all of the generic agent functionality.

To trace TCP parameters, mark each parameter with ‘‘$tcp trace window_’’ and then send the output to a trace file with ‘‘$tcp attach [open trace.tr w]’’.

Tcp segments can be sent with the advance and advanaceby commands. When all data is sent, the done method will be invoked (which can be overridden in OTcl).

$tcp advance n
Send up to the nth packets.

$tcp advanceby n
Send n more packets.

$tcp done
Functional called when all packets (specified by advance/advanceby/maxpkts_) have been sent. Can be overriden on a per-object basis.

Configuration Parameters

*window_*
The upper bound on the advertised window for the TCP connection (in packets).

*maxcwnd_*
The upper bound on the congestion window for the TCP connection. Set to zero to ignore. (This is the default.) Measured in packets.

*windowInit_*
The initial size of the congestion window on slow-start. (in packets).

*wnd_init_option_*
The algorithm used for determining the initial size of the congestion window. Set to 1 for a static algorithm using the value in *windowInit_*. Set to 2 for a dynamic algorithm using a function of *packetSize_*.

*syn_*
Set to true to model the initial SYN/ACK exchange in one-way TCP. Set to false as default.

*delay_growth_*
Set to true to delay the initial congestion window until after one packet has been sent and acked. Set to false as default.

*windowOption_*
The algorithm to use for managing the congestion window in linear phase. The standard algorithm is 1 (the default). Other experimental algorithms are documented in the source code.

*windowThresh_*
Gain constant to exponential averaging filter used to compute *awnd* (see below). For investigations of different window-increase algorithms.

*overhead_*
The range (in seconds) 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. This allows a Fast Retransmit after a quench() due to an ECN (explicit congestion notification) bit.

*packetSize_*
The size in bytes to use for all packets from this source.

*tcpip_base_hdr_size_*
The size in bytes of the base TCP/IP header.

*tcpTick_*
The TCP clock granularity for measuring roundtrip times. Note that it is set by default to the non-standard value of 100ms. Measured in seconds.

*bugFix_*
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.

*slow_start_restart_*
Boolean; set to 1 to slow-start after the connection goes idle. On by default.

*srtt_init_*
Initial value for the smoothed roundtrip time estimate. Default is 0 seconds.

*t_rttvar_*
Initial value for the variance in roundtrip time. Default is 3 seconds.

*rtxcur_init_*
Initial value for the retransmit value. Default is 6 seconds.

T_SRTT_BITS
Exponent of weight for updating the smoothed round-trip time t_srtt_. Default is 3, for a weight of 1/2ˆT_SRTT_BITS or 1/8.

T_RTTVAR_BITS
Exponent of weight for updating variance in round-trip time, t_rttvar_. Default is 2, for a weight of 1/2ˆT_RTTVAR_BITS or 1/4.

*rttvar_exp_*
Exponent of multiple of the mean deviation in calculating the current retransmit value t_rtxcur_. Default is 2, for a multiple of 2ˆrttvar_exp_ or 4.

Defined Constants

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.

State Variables

*dupacks_*
Number of duplicate acks seen since any new data was acknowledged.

*seqno_* Highest sequence number for data from data source to TCP.

*t_seqno_*
Current transmit sequence number.

*ack_* Highest acknowledgment seen from receiver.

*cwnd_* Current value of the congestion window (in packets).

*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 (in packets).

*rtt_* Round-trip time estimate. In seconds (expressed in multiples of tcpTick_).

*srtt_* Smoothed round-trip time estimate. In seconds (in multiples of tcpTick_/8).

*rttvar_* Round-trip time mean deviation estimate.

*t_rtxcur_*
Current retransmit value. In seconds.

*backoff_*
Round-trip time exponential backoff constant.

TCP/RENO OBJECTS

TCP/Reno objects are a subclass of TCP objects that implement the Reno TCP transport protocol as described in [7]. There are no methods, configuration parameters or state variables specific to this object.

TCP/NEWRENO OBJECTS

TCP/Newreno objects are a subclass of TCP objects that implement a modified version of the BSD Reno TCP transport protocol.

There are no methods or state variables specific to this object.

Configuration Parameters

*newreno_changes_*
Set to zero for the default NewReno described in [7]. Set to 1 for additional NewReno algorithms as suggested in [10]; this includes the estimation of the ssthresh parameter during slow-start.

TCP/VEGAS OBJECTS

This section of the man page has not yet been written.

TCP/SACK1 OBJECTS

TCP/Sack1 objects are a subclass of TCP objects that implement the BSD Reno TCP transport protocol with Selective Acknowledgement Extensions as described in [7].

They inherit all of the TCP object functionality. There are no methods, configuration parameters or state variables specific to this object.

TCP/FACK OBJECTS

TCP/Fack objects are a subclass of TCP objects that implement the BSD Reno TCP transport protocol with Forward Acknowledgement congestion control.

They inherit all of the TCP object functionality. There are no methods or state variables specific to this object.

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)

TCP/FULLTCP OBJECTS

This section has not yet been added to the man page. The implementation and the configuration parameters are described in [11].

TCPSINK OBJECTS

TCPSink 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. TCPSink objects inherit all of the generic agent functionality. There are no methods or state variables specific to the TCPSink object.

Configuration Parameters

*packetSize_*
The size in bytes to use for all acknowledgment packets.

*maxSackBlocks_*
The maximum number of blocks of data that can be acknowledged in a SACK option. For a receiver that is also using the time stamp option [RFC 1323], the SACK option specified in RFC 2018 has room to include three SACK blocks. This is only used by the TCPSink/Sack1 subclass. This value may not be increased within any particular TCPSink object after that object has been allocated. (Once a TCPSink object has been allocated, the value of this parameter may be decreased but not increased).

TCPSINK/DELACK OBJECTS

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

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.

TCPSINK/SACK1 OBJECTS

TCPSink/Sack1 objects are a subclass of TCPSink that implement a SACK receiver for TCP packets. They inherit all of the TCPSink object functionality. There are no methods, configuration parameters or state variables specific to this object.

TCPSINK/SACK1/DELACK OBJECTS

TCPSink/Sack1/DelAck objects are a subclass of TCPSink/Sack1 that implement a delayed-SACK receiver for TCP packets. They inherit all of the TCPSink/Sack1 object functionality. There are no methods or state variables specific to this object.

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.

SRM OBJECTS

SRM objects are a subclass of agent objects that implement the SRM reliable multicast transport protocol. They inherit all of the generic agent functionalities.

$srm traffic-source *source*
Attach a traffic source, e.g., Application/Traffic/CBR, to the SRM agent.

$srm start
Join the multicast group, start the SRM agent and its attached traffic source.

$srm delete
Stop the SRM agent, delete all its status and detach the traffic source.

$srm trace *trace-file*
Write the traces generated by the SRM agent to *trace-file*. The traces includes timer settings, request and repair sending and receipts, etc. Two related files that are not built into ns are *tcl/mcast/srm-debug.tcl* that permits more detailed tracing of the delay computation functions, and *tcl/mcast/srm-nam.tcl* that separately marks srm control messages from data. The latter is useful to enhance nam visualisation.

$srm log *log-file*
Write the recovery statistics during each request or repair to *log-file*. The statistics include start time, duration, message id, total number of duplicate requests and repairs.

$srm distance? *node*
Return the distance estimate to *node* in this SRM agent.

$srm distances? *node*
Returns a list of <group member, distance> tuples of the distances to all group members that this node is aware of. The group member is identified as the address of the remote agent. The first tuple is this agent's token. The list can be directly loaded into a Tcl array.

Configuration Parameters

*packetSize_*
The data packet size in bytes that will be used for repair messages. The default value is 1024.

*requestFunction_*
The algorithm used to produce a retransmission request, e.g., setting request timers. The default value is SRM/request. Other possible request functions are SRM/request/Adaptive, used by the Adaptive SRM code.

*repairFunction_*
The algorithm used to produce a repair, e.g., compute repair timers. The default value is SRM/repair. Other possible request functions are SRM/repair/Adaptive, used by the Adaptive SRM code.

*sessionFunction_*
The algorithm used to generate session messages. Default is SRM/session

*sessionDelay_*
The basic interval of session messages. Slight random variation is added to this interval to avoid global synchronization of session messages. User may want to adjust this variable according to their specific simulation. Measured in seconds; default value is 1.0 seconds.

*C1_, C2_*
The parameters which control the request timer. Refer to [8] for detail. The default value is *C1_* = *C2_* = 2.0.

*D1_, D2_*
The parameters which control the repair timer. Refer to [8] for detail. The default value is *D1_* = *D2_* = 1.0.

*requestBackoffLimit_*
The maximum number of exponential backoffs. Default value is 5.

State Variables

*stats_* An array containing multiple statistics needed by adaptive SRM agent. Including: duplicate requests and repairs in current request/repair period, average number of duplicate requests and repairs, request and repair delay in current request/repair period, average request and repair delay.

SRM/Adaptive OBJECTS

SRM/Adaptive objects are a subclass of the SRM objects that implement the adaptive SRM reliable multicast transport protocol. They inherit all of the SRM object functionalities.

State Variables Refer to the SRM paper by Sally et al ([11]) for more detail.

*pdistance_*
This variable is used to pass the distance estimate provided by the remote agent in a request or repair message.

*D1_, D2_*
The same as that in SRM agents, except that they are initialized to log10(group size) when generating the first repair.

*MinC1_, MaxC1_, MinC2_, MaxC2_*
The minimum/maximum values of C1_ and C2_. Default initial values are defined in [8]. These values define the dynamic range of *C1_* and *C2_*.

*MinD1_, MaxD1_, MinD2_, MaxD2_*
The minimum/maximum values of D1_ and D2_. Default initial values are defined in [8]. These values define the dynamic range of *D1_* and *D2_*.

AveDups
Higher bound for average duplicates.

AveDelay
Higher bound for average delay.

eps *AveDups - dups* determines the lower bound of the number of duplicates, when we should adjust parameters to decrease delay.

APPLICATION OBJECTS

Application objects generate data for transport agents to send.

FTP APPLICATION OBJECTS

Application/FTP objects produce bulk data for a TCP object to send.

$ftp start
Causes FTP to produce packets indefinitely.

$ftp produce *n*
Causes the FTP object to produce *n* packets instantaneously.

$ftp stop
Causes the attached TCP object to stop sending data.

$ftp attach *agent*
Attaches an Application/FTP object to *agent*.

$ftp producemore *count*
Causes the Application/FTP object to produce *count* more packets.

Configuration Parameters

maxpkts
The maximum number of packets generated.

TELNET APPLICATION OBJECTS

Application/Telnet objects produce individual packets with inter-arrival times as follows. If *interval_* is non-zero, then inter-arrival times are chosen from an exponential distribution with average *interval_*. If *interval_* is zero, then inter-arrival times are chosen using the "tcplib" telnet distribution.

$telnet start
Causes the Application/Telnet object to start producing packets.

$telnet stop
Causes the Application/Telnet object to stop producing packets.

$telnet attach *agent*
Attaches a Application/Telnet object to *agent*.

Configuration Parameters

*interval_*
The average inter-arrival time in seconds for packets generated by the Application/Telnet object.

TRAFFIC OBJECTS

Traffic objects create data for a transport protocol to send. A Traffic object is created by instantiating an object of class Application/Traffic/*type* where *type* is one of Exponential, Pareto, CBR, Trace.

EXPONENTIAL TRAFFIC OBJECTS

Application/Traffic/Exponential objects generate On/Off traffic. During "on" periods, packets are generated at a constant burst rate. During "off" periods, no traffic is generated. Burst times and idle times are taken from exponential distributions.

Configuration Parameters

*packet_size_*
The packet size in bytes.

*burst_time_*
Burst duration in seconds.

*idle_time_*
Idle time in seconds.

*rate_* Peak rate in bits per second.

PARETO TRAFFIC OBJECTS

Application/Traffic/Pareto objects generate On/Off traffic with burst times and idle times taken from pareto distributions.

Configuration Parameters

*packet_size_*
The packet size in bytes.

*burst_time_*
Average on time in seconds.

*idle_time_*
Average off time in seconds.

*rate_* Peak rate in bits per second.

*shape_* Pareto shape parameter.

CBR (CONSTANT BIT RATE) TRAFFIC OBJECTS

Application/Traffic/CBR objects generate packets at a constant rate. Dither can be added to the interarrival times by enabling the "random" flag.

Configuration Parameters

*rate_* Peak rate in bits per second.

*packet_size_*
The packet size in bytes.

*random_*
Flag that turns dithering on and off (default is off).

*maxpkts_*
Maximum number of packets to send.

TRACE TRAFFIC OBJECTS

Application/Traffic/Trace objects are used to generate traffic from a trace file.

$trace attach-tracefile *tfile*
Attach the Tracefile object *tfile* to this trace. The Tracefile object specifies the trace file from which the traffic data is to be read (see TRACEFILE OBJECTS section). Multiple Application/Traffic/Trace objects can be attached to the same Tracefile object. A random starting place within the Tracefile is chosen for each Application/Traffic/Trace object.

There are no configuration parameters for this object.

TRACEFILE OBJECTS

Tracefile objects are used to specify the trace file that is to be used for generating traffic (see TRAFFIC/TRACE OBJECTS section). $tracefile is an instance of the Tracefile Object.

$tracefile filename *trace-input*
Set the filename from which the traffic trace data is to be read to *trace-input.*

There are no configuration parameters for this object. A trace file consists of any number of fixed length records. Each record consists of 2 32 bit fields. The first indicates the interval until the next packet is generated in microseconds. The second indicates the length of the next packet in bytes.

TRACE AND MONITORING METHODS

[NOTE: This section has not been verified to be up-to-date with the release.] Trace objects are used to generate event level capture logs typically to an output file. Throughout this section $ns refers to a Simulator object, $agent refers to an Agent object.

$ns create-trace *type fileID node1 node2 [option]*

Create a Trace object of type *type* and attach the filehandle *fileID* to it to monitor the queues between nodes *node1* and *node2. type* can be one of Enque, Deque, Drop. Enque monitors packet arrival at a queue. Deque monitors packet departure at a queue. Drop monitors packet drops at a queue. *fileID* must be a file handle returned by the Tcl *open* command and it must have been opened for writing. If *option* is not specified, the command will instruct the created trace object to generate ns traces. If *option* is """nam""" the new object will produce nam traces. Returns a handle to the trace object.

$ns drop-trace *node1 node2 trace*

Remove trace object attached to the link between nodes *node1* and *node2* with *trace* as the object handle.

$ns trace-queue *node1 node2 fileID*

Enable Enque, Deque and Drop tracing on the link between *node1* and *node2.*

$ns namtrace-queue *node1 node2 fileID*

Same function as **$ns trace-queue**, except it produces nam traces.

$ns trace-all *fileID*

Enable Enque, Deque, Drop Tracing on all the links in the topology created after this method is invoked. Also enables the tracing of network dynamics. *fileID* must be a file handle returned by the Tcl *open* command and it must have been opened for writing.

$ns namtrace-all *fileID*

Same function as **$ns trace-all**, except it will produce all equivalent traces in nam format. In addition, calling this command *before* the simulator starts to run will generate color configurations (if any) and topology information needed by nam (nodes, links, queues). An example can be found at ns-2/tcl/ex/nam-example.tcl.

$ns namtrace-config *fileID*

Assign a file to store nam configuration information, e.g., node/link/agents and some Simulator-related traces such as annotations. When you don't want to trace every object. call this function and then use *$ns namtrace-queue, rtModel trace,* etc., to insert traces individually. Note that you should use the same file for individual traces and nam configuration. An example for this is available at ns-2/tcl/ex/nam-separate-trace.tcl.

$ns monitor-queue *node1 node2*

Arrange for queue length of link between nodes *node1* and *node2* to be tracked. Returns QueueMonitor object that can be queried to learn average queue size etc. [see QueueMonitor Objects section]

$ns flush-trace

Flush the output channels attached to all the trace objects.

$link trace-dynamics *ns fileID [option]*

Trace the dynamics of this link and write the output to *fileID* filehandle. *ns* is an instance of the Simulator or MultiSim object that was created to invoke the simulation.

$ns color *id name*

Create a color index, which links the number *id* to the color name *name*. All colors created *before* the simulator starts to run will be written to nam trace file, if there is any.

$ns trace-annotate *string*

Writes an annotation to ns and nam trace file, if there are any. The string should be enclosed in double quote to make it a single argument.

trace_annotate *string*

Another version of **$ns trace-annotate**, which is a global function and doesn't require the caller to know ns.

$ns duplex-link-op $node1 $node2 $op $args

Perform a given operation *$op* on the given duplex link (*$node1*, *$node2*). The following two operations may be used:

orient	- Specify the nam orientation of the duplex link. Values can be left, right, up, down, their mixture combined by '-' (e.g., left-down), and a number specifying the angle between the link and the horizontal line.
queuePos	- Construct a queue of the simplex link (*$node1*, *$node2*) in nam, and specify the angle between the horizontal line and the line along which the queued packets will be displayed.

$ns add-agent-trace *agent name [fileID]*

Write a nam trace line, which will create a trace agent for *agent* when interpreted by nam. The trace agent's name will be *name*. This nam trace agent is used to show the position of *agent* and can be used to write nam traces of variables associated with the agent. By default traces will be written to the file assigned by *namtrace-all*. *fileID* can be used to write traces to another file.

$agent tracevar *name*

Label OTcl variable *name* of **$agent** to be traced. Then whenever the variable *name* changes value, a nam trace line will be written to nam trace file, if there is one. Note that *name* must be the same as the variable's real OTcl name.

$ns delete-agent-trace *agent*

Write a nam trace line, which will delete the nam trace associated with *agent* when interpreted by nam.

$agent add-var-trace *name value [type]*

Write a nam trace line, which creates a variable trace with name *name* and value *value*, when interpreted by nam. *type* indicates the type of the variable, e.g., is it a list, array, or a plain variable. Currently only plain variable is supported, for which *type* = 'v'.

The following 2 functions should be called *after* the simulator starts running. This can be done using **$ns at**.

$agent delete-var-trace *name*

Write a nam trace line, which deletes the variable trace *name* when interpreted by nam.

$agent update-var-trace *name value [type]*

Write a nam trace line, which changes the value of traced variable *name* when interpreted by nam. Unlike **$agent tracevar**, the above 3 functions provide ’manual’ variable tracing, in which variable tracing are done by placing **$agent update-var-trace** in OTcl code, while *tracevar* automatically generates nam traces when the traced variable changes value.

The tracefile format is backward compatible with the output files in the ns version 1 simulator so that ns-1 post-processing scripts can still be used. Trace records of traffic for link objects with Enque, Deque or Drop Tracing have the following form:

```
<code> <time> <hsrc> <hdst> <packet>
```

where

```
<code> := [hd+-r] h=hop d=drop +=enque -=deque r=receive
<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> <flowID> <src.sport> <dst.dport> <seq> <pktID>
<type> := tcp|telnet|cbr|ack etc.
<size> := packet size in bytes
<flags> := [CP]  C=congestion, P=priority
<flowID> := flow identifier field as defined for IPv6
<src.sport> := transport address (src=node,sport=agent)
<dst.sport> := transport address (dst=node,dport=agent)
<seq> := packet sequence number
<pktID> := unique identifer for every new packet
```

Only those agents interested in providing sequencing will generate sequence numbers and hence this field may not be useful for packets generated by some agents.

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
```

Trace records for link dynamics are of the form:

```
<code> <time> <state> <src> <dst>
```

where

```
<code> := [v]
<time> := simulation time in seconds
<state> := [link-up | link-down]
<src> := first node address of link
<dst> := second node address of link
```

INTEGRATOR Objects

Integrator Objects support the approximate computation of continuous integrals using discrete sums. The running sum(integral) is computed as: sum_ += [lasty_ * (x - lastx_)] where (x, y) is the last element entered and (lastx_, lasty_) was the element previous to that added to the sum. lastx_ and lasty_ are updated as new elements are added. The first sample point defaults to (0,0) that can be changed by changing the values of (lastx_,lasty_).

$integrator newpoint *x y*
Add the point (x,y) to the sum. Note that it does not make sense for x to be less than lastx_.

There are no configuration parameters specific to this object.

State Variables

*lastx_*	x-coordinate of the last sample point.
*lasty_*	y-coordinate of the last sample point.
*sum_*	Running sum (i.e. the integral) of the sample points.

SAMPLES Objects

Samples Objects support the computation of mean and variance statistics for a given sample.

$samples mean
Returns mean of the sample.

$samples variance
Returns variance of the sample.

$samples cnt
Returns a count of the sample points considered.

$samples reset
Reset the Samples object to monitor a fresh set of samples.

There are no configuration parameters or state variables specific to this object.

BUILTINS

[NOTE: This section has not been verified to be up-to-date with the release.] Because *OTcl* is a full-fledged programming language, it is easy to build high-level simulation constructs from the ns primitives. Several library routines have been built in this way, and are embedded into the ns interpreter as methods of the Simulator class. Throughout this section $ns represents a Simulator object.

$ns create-connection *srcType srcNode dstType dstNode class*
Create a source agent of type *srcType* at node *srcNode* and connect it to a destination agent of type *dstType* at node *dstNode.* Also, connect the destination agent to the source agent. The traffic class of both agents is set to *class.* This method returns the source agent.

EXAMPLE

```
set ns [new Simulator]

#
# Create two nodes
#
set n0 [$ns node]
set n1 [$ns node]

#
# Create a trace and arrange for all the trace events of the
# links subsequently created to be dumped to "out.tr"
#
set f [open out.tr w]
$ns trace-all $f
```

```
#
# Connect the two nodes with a 1.5Mb link with a transmission
# delay of 10ms using FIFO drop-tail queuing
#
$ns duplex-link $n0 $n1 1.5Mb 10ms DropTail

#
# Set up BSD Tahoe TCP connections in opposite directions.
#
set tcp_src1 [new Agent/TCP]
set tcp_snk1 [new Agent/TCPSink]
set tcp_src2 [new Agent/TCP]
set tcp_snk2 [new Agent/TCPSink]
$ns attach-agent $n0 $tcp_src1
$ns attach-agent $n1 $tcp_snk1
$ns attach-agent $n1 $tcp_src2
$ns attach-agent $n0 $tcp_snk2
$ns connect $tcp_src1 $tcp_snk1
$ns connect $tcp_src2 $tcp_snk2

#
# Create ftp sources at the each node
#
set ftp1 [$tcp_src1 attach-source FTP]
set ftp2 [$tcp_src2 attach-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"

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

DEBUGGING

To enable debugging when building ns from source:

```
% ./configure --enable-debug
% make
```

For more details about ns debugging please see <http://www-mash.cs.berkeley.edu/ns/ns-debugging.html>.

DIFFERENCES FROM NS-1

In general, more complex objects in ns-1 have been broken down into simpler components for greater flexibility and composability. Details of differences between ns-1 and ns-2 can be found at <http://www-mash.cs.berkeley.edu/ns/ns.html>.

HISTORY

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/.

SEE ALSO

Tcl(1), tclsh(1), nam(1), otclsh

[1] S. Keshav, ‘‘REAL: A Network Simulator’’. UCB CS Tech Report 88/472, December 1988. See http://minnie.cs.adfa.oz.au/REAL/index.html for more information.

[2] Floyd, S. and Jacobson, V. Random Early Detection gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, Vol. 1, No. 4. August 1993. pp. 397-413. Available from http://www-nrg.ee.lbl.gov/floyd/red.html.

[3] Floyd, S. Simulator Tests. July 1995. URL ftp://ftp.ee.lbl.gov/papers/simtests.ps.Z.

[4] 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.

[5] Floyd, S., and Jacobson, V. Link-sharing and Resource Management Models for Packet Networks. IEEE/ACM Transactions on Networking, Vol. 3 No. 4, August 1995. pp. 365-386.

[6] Floyd, S., Notes of Class-Based Queueing: Setting Parameters. URL ftp://ftp.ee.lbl.gov/papers/params.ps.Z. September 1995.

[7] Fall, K., and Floyd, S. Comparisons of Tahoe, Reno, and Sack TCP. December 1995. URL ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z.

[8] David Wetherall and Christopher J. Linblad. Extending Tcl for Dynamic Object-Oriented Programming. In Proceedings of the USENIX Tcl/Tk Workshop, Toronto, Ontario, USENIX. July, 1995. At <http://www.tns.lcs.mit.edu/publications/tcltk95.djw.html>.

[9] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In Proc. of SIGCOMM, pp. 231-242, 1995.

[10] Hoe, J., Improving the Start-up Behavior of a Congestion Control Scheme for TCP. in SIGCOMM 96, August 1996, pp. 270-280. URL http://www.acm.org/sigcomm/sigcomm96/papers/hoe.html.

[11] Fall, K., Floyd, S., and Henderson, T., Ns Simulator Tests for Reno FullTCP. URL ftp://ftp.ee.lbl.gov/papers/fulltcp.ps. July 1997.

[12] Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S. and Zhang, L., A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. To appear in IEEE/ACK Transaction on Networking, November 1996. ftp://ftp.ee.lbl.gov/papers/srm1.ps.gz

[13] Fall, K., and Varadhan, K., (eds.), "Ns notes and documentation", work in progress. http://www-mash.cs.berkeley.edu/ns/nsDoc.ps.gz

Research using ns is on-going. A list of recent research contributions employing ns can be found at <http://www-mash.cs.berkeley.edu/ns/ns-research.html>.

Work on ns is on-going. Information about the most recent version is available at <http://www-mash.cs.berkeley.edu/ns/ns.html>.

A mailing list for ns users and announcements is also available, send mail to ns-users-request@mash.cs.berkeley.edu or ns-announce-request@mash.cs.berkeley.edu to join. Questions should be forwarded to ns-users; ns-announce will be low-traffic announcements only.

AUTHORS

Steven McCanne (mccanne@ee.lbl.gov), University of California, Berkeley and Lawrence Berkeley National Laboratory, Berkeley, CA, and Sally Floyd (floyd@ee.lbl.gov) Lawrence Berkeley National Laboratory, Berkeley, CA. A complete list of contributors to ns is at <http://www-mash.cs.berkeley.edu/ns/ns-contributors.html>.

BUGS

Not all of the functionality supported in ns-1 has been ported to ns-2.

This manual page is incomplete.