

# QNS User's Manual

Created by: Socrates Toussas, Ala Khanbashi & Dincer Terzioglu

3rd November 2003

### **Abstract**

The User's Manual for the Queueing Network Simulator, created by Rami Mukhtar (University of Melbourne), describes all the operational aspects of the QNS and how a new user would go about utilising and extending it by creating new network modules.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>How to Use the Queuing Network Simulator</b>	<b>5</b>
<b>3</b>	<b>Existing Network Entities</b>	<b>6</b>
3.1	Links . . . . .	6
3.2	Traffic Sources . . . . .	6
3.3	Transmit Processes . . . . .	6
3.4	Flow Control Systems . . . . .	7
3.5	Routers . . . . .	7
<b>4</b>	<b>Roles of Existing Network Entities in Simulations</b>	<b>7</b>
<b>5</b>	<b>Three Stages of a Simulation</b>	<b>8</b>
5.1	Parsing of World File and Construction of Network . . . . .	8
5.2	Executing the Simulation (Processing the Event Queue) . . . . .	9
<b>6</b>	<b>World File</b>	<b>10</b>
<b>7</b>	<b>How to Create a World File</b>	<b>10</b>
7.1	ID numbers . . . . .	10
7.2	Define all Traffic Sources and Transmit Processes . . . . .	11
7.2.1	Traffic Source . . . . .	11
7.2.2	Transmit Process . . . . .	11
7.3	Define all nodes . . . . .	12
7.3.1	Links (Link, SLink) . . . . .	12
7.3.2	Flow Control Systems (Generator, TCP, SFC_Rec) . . . . .	13
7.3.3	Routers . . . . .	14
7.4	Define all connections . . . . .	14
7.5	List the events . . . . .	16
7.5.1	One-off events (START, STOP and LOG) . . . . .	16
7.5.2	Periodic Events (LOG) . . . . .	17
7.6	Specify the time at which simulation should end . . . . .	17
<b>8</b>	<b>Construction of an Example World</b>	<b>17</b>
8.1	Complete World File . . . . .	20
<b>9</b>	<b>How to Add a New Module</b>	<b>21</b>
9.1	The Constructor . . . . .	22
9.2	Required Functions for Routers, Generators and Links . . . . .	22
9.2.1	Functions to service event START or STOP . . . . .	22
9.2.2	Function to service event PASS . . . . .	22
9.2.3	Function to service event SERVICE . . . . .	23
9.2.4	Function to service event LOG . . . . .	23
9.3	Required Functions for Traffic Sources . . . . .	23

9.4	Required Functions for Transmit Processes . . . . .	24
<b>10</b>	<b>Creating a Module of Type Router</b>	<b>24</b>
10.1	Create a .h file . . . . .	24
10.2	Create a .cpp file . . . . .	26
10.3	Modify main.h . . . . .	28
10.4	Modify main.cpp . . . . .	29
10.5	Modify the Makefile . . . . .	30
10.6	Note the changes to the World File syntax . . . . .	30
<b>11</b>	<b>Creating a Module of Type Generator</b>	<b>30</b>
11.1	Create a .h file . . . . .	31
11.2	Create a .cpp file . . . . .	33
11.3	Modify main.h . . . . .	35
11.4	Modify main.cpp . . . . .	35
11.5	Modify the Makefile . . . . .	37
11.6	Note the changes to the World File syntax . . . . .	37
<b>12</b>	<b>Creating a Module of Type Link</b>	<b>37</b>
12.1	Create a .h file . . . . .	38
12.2	Create a .cpp file . . . . .	39
12.3	Modify main.h . . . . .	41
12.4	Modify main.cpp . . . . .	42
12.5	Modify the Makefile . . . . .	43
12.6	Note the changes to the World File syntax . . . . .	43
<b>13</b>	<b>Creating a Module of Type Traffic Source</b>	<b>43</b>
13.1	Create a .h file . . . . .	43
13.2	Create a .cpp file . . . . .	45
13.3	Modify main.h . . . . .	47
13.4	Modify main.cpp . . . . .	47
13.5	Modify the Makefile . . . . .	48
13.6	Note the changes to the World File syntax . . . . .	48
<b>14</b>	<b>Creating a Module of Type Transmit Process</b>	<b>49</b>
14.1	Create a .h file . . . . .	49
14.2	Create a .cpp file . . . . .	51
14.3	Modify main.h . . . . .	52
14.4	Modify main.cpp . . . . .	52
14.5	Modify the Makefile . . . . .	53
14.6	Note the changes to the World File syntax . . . . .	53

## 1 Introduction

The QNS simulates packet flow through a user created network. It achieves this by maintaining a priority queue of events that are scheduled by the entities that form the network. The events themselves represent basic actions that occur in a network, such as a packet being transferred from one node to another or a node that starts or stops operating. The actual simulation occurs when the events in the priority queue are serviced individually. Servicing one event presents the possibility that other events will be scheduled at a later time as a result of this event. This phenomenon mainly occurs when a packet repeatedly passes from one node to another until it reaches it's destination. A chain reaction of events is scheduled (and inserted into the priority queue) to model the packet's journey. The QNS does not concern itself with the contents of the packets but rather their size.

For the QNS to succeed in performing packet flow simulations, the user must provide a textual representation of the topology of the network (in the form of a World File - see Section 6). The QNS examines the contents of the file and creates an internal representation of the same network. This internal representation consists of node objects (e.g. Routers, Links, Traffic Sources, Generators), Transmit Process objects (that model packet transmission over a stochastic link), a Node Map (that keeps track of the network entities) and a Connection Table (that keeps track of the connections between the nodes). In the World File, the user must also specify the events that will initiate the simulation. These events determine when the nodes in the network will start operating and begin scheduling other events. With the network created and the initial events scheduled, the only remaining task is the execution of the event queue (see Section 5.2). This is achieved by traversing the event queue and servicing each event separately.

## 2 How to Use the Queueing Network Simulator

The process of using the QNS to construct and simulate a network is extremely simple. All that is required of the user is to become familiar with the existing objects that can be used to create a network (see Existing Network Entities in Section 3)), and then creating a textual representation of the topology of that network to enable it's simulation. This textual representation must be written in a file that strictly conforms to a standard format.

This file is referred to as the "World File", as it contains all the specifications of the entities found in the "world" (in this case, the network). See the sections World File (Section 6) and How to create a World File (Section 7). The final step in the process is to actually simulate the network. This is accomplished on the command prompt by typing the following line:

```
./qns world_file
```

where:

**qns:** is the name of the executable that starts the simulator.

*world\_file*: is the name of the World File that holds the information for the network you are simulating.

Before the simulation begins, parsing of the World File and construction of the network in memory must occur(see Section 5.1). Various messages describing this process are printed to standard output. If logging has been specified in the World File, lines of statistics are then printed to standard output. Finally, a message specifying that the simulation has completed successfully is then printed to standard output, together with the total number of events that were executed.

### 3 Existing Network Entities

In this version of the QNS, there a number of entities which can be utilised to construct networks that are representative of those found in the real world. A brief description of these entities is provided below.

#### 3.1 Links

These are uni-directional point to point connections between two nodes in the network. They can either be of type "Link", which is a standard connection, or of type "SLink" which is a connection with stochastic properties.

#### 3.2 Traffic Sources

Traffic Sources create the packets of data that flow through a network. A Traffic Source can be thought of as a computer running an application which creates data to communicate with another node in the network.

There are two types of Traffic Sources that can be included in a simulation: one is "TS\_Greedy", which continually queues packets to be sent as soon as they are ready, and the other is "TS\_Exp", which outputs packets at an exponential distributed offset time after they are actually created. A node of type "Generator" or TCP (see Section 3.4 below), which transmits packets through a network, must have its own Traffic Source to provide it with the packets it requires. These types of nodes can also have the sole function of absorbing packets and therefore do not require an accompanying Traffic Source.

#### 3.3 Transmit Processes

A Transmit Process is an object which models the transmission of packets over a medium (a stochastic link - SLink). There are two types of Transmit Process defined. One is "TP\_general" which is a general packet transmission model. The other is "TP\_two\_state" which simulates a two state Markov channel transmission process. See the QNS Reference Manual for more information on Transmit Processes.

### 3.4 Flow Control Systems

These are nodes that generate and/or absorb packets using window flow control.

A Generator uses the simplest form of flow control, where the window size is fixed and does not change according to any algorithm.

A TCP node applies flow control algorithms which change the window size.

A Stochastic Flow Control (SFC) Filter (defined as SFC\_Rec in the source code) sits between a Link and a receiver (conceptually the same as a Router) and utilises the extra information stored, by a SFC Router, in the headers of SFC packets to adjust the window size. Other types of packets are treated as per normal, hence why it is considered a filter.

See the QNS Reference Manual for more information on these types of nodes.

### 3.5 Routers

The Router node provides the basic functionality of a router. Each output port of the Router has its own buffer (packet queue) and incoming packets are placed in the buffer into the corresponding buffer.

Another Router available for simulation is the SFC Router. It is an adaptation of a normal router to enable the support of the SFC protocol (as developed by CUBIN, University of Melbourne).

**For a more thorough description of all these objects and their internal operations, refer to the QNS Reference Manual.**

## 4 Roles of Existing Network Entities in Simulations

To simulate packet flow, packets must be present in a network. A Traffic Source creates new data packets and passes them onto a Generator (or TCP node). The Generator inserts (into the network) packets that must travel to their destination nodes (which are also Generators or TCP nodes), where they are received and consequently removed from the network.

Each packet reaches its destination via uni-directional Links (regular point-to-point links) and SLinks (stochastic transmission links), as well as various types of Routers that calculate the cheapest path that a packet must take to get there. Once there, the Generator produces an acknowledgement (packet) that returns to the source node to indicate that the data packet was successfully received (thus modelling a reliable service). An acknowledgement adds to the network traffic as it is treated as a regular packet by the system. The number of packets in the network at any one time depends on the size of the send and receive windows of all the Flow Control Systems in the network.

Transmit Processes attempt to emulate the transmission of packets over the Data Link Layer of a network, which is concerned only with the transmission of data across each link separately. They model the process of breaking down the

packets into frames and transmitting them down Links that exhibit stochastic properties - SLinks.

## 5 Three Stages of a Simulation

Each simulation is performed by the QNS in three main stages:

- Parsing of the World File.
- Creation of the network topology specified in the World File.
- Execution of the event queue.

### 5.1 Parsing of World File and Construction of Network

Parsing of the World File occurs in conjunction with the creation of the network. A World File contains lines of text that specify the network entities (Routers, Links, etc.) that form the topology to be simulated. Refer to Existing Network Entities (Section 3), World File (Section 6) and How to Create a World File (Section 7) for more information.

Each line of text is parsed into the main module of the simulator to create the object that represents the relevant network entity. After the object has been constructed, a pointer to the object is inserted into a structure called the "Node\_map" (this is true for all node objects only. Transmit Process objects are placed in a separate map container). The Node Map is essentially a table of pairs (node identification (id) number, pointer to the node) that provides a link between the node id and the actual object in memory. See the QNS Reference Manual for more information on the Node\_map class.

A World File must also include lines of text that define the connections between nodes in the network. These lines are also parsed into the main module where they are then converted into "Con\_table" entries. The Con\_table (or "Connection Table") is basically a list of connections between nodes where each entry consists of the source node id, the destination node id and the cost of travelling from one to the other. Both structures are utilised by Routers to construct their routing tables. See the QNS Reference Manual for more information on the Con\_table class.

Each World File must also contain lines of text that define events of type START (see Section 5.2) and STOP (see Section 5.2). These events specify the times at which the nodes in the network will begin and end their operation (when not operating, nodes cannot send or receive packets). Events of type LOG (see Section 5.2), that enable real time analysis of events occurring at a node, can also be included. These lines are parsed into the main module which creates the relevant event objects and inserts them into the event queue. When execution of the event queue begins, the initial START events trigger the continual creation of other events for the duration of the simulation (the duration is specified on the final line of the world file).

## 5.2 Executing the Simulation (Processing the Event Queue)

As with many common computer based simulators, the QNS maintains an ordered structure of event objects that represents the sequence of events that occur during a simulation. A priority queue that sorts events in order of increasing time of occurrence is the structure of choice. The priority queue is serviced, one event at a time, until the prescribed duration has expired. For each event, the handler is invoked to delegate which functions the node should execute to service the event. See the QNS Reference Manual for more information on events.

A PASS event simulates a packet moving from one node (the source node that scheduled the event) to another node (the destination node).<sup>1</sup> If the destination node is busy, the event is re-inserted into the queue to be serviced at a later time (i.e. the packet must wait until the node is free). If the destination node is free, the source node's `out_event()` function is called (if `out_event()` is not implemented, `out()` is called instead) to obtain the packet that will be passed to the destination node's `in()` function. The execution of the `in()` function will generate a new PASS event that will be inserted into the event queue and serviced when the packet is ready to proceed to the next node. This process is repeated until the packet reaches its destination. Each particular PASS event can only occur once, therefore it is deleted immediately after it has been serviced.

A START event simulates a node being brought online to begin operation, i.e. to begin sending/receiving packets. If the node has a `start()` function defined by the user, it is executed, otherwise the default `start()` function is executed. This type of event is not periodic (it does not continuously occur) and is therefore deleted after it has been serviced.

A STOP event simulates a node being taken off-line to cease its operation, i.e. to stop sending/receiving packets. The default `stop()` function is called, unless the user has provided their own, in which case this function is called. This type of event is also not periodic, and is therefore deleted after it has been serviced.

A LOG event enables the user to observe various statistics and attributes of a node during a simulation. It is a service provided to the user for analytical and diagnostic purposes and to indicate the sequence of events as they occur (e.g. a packet arriving at node *n* at time *t*). The `log()` function of a node is called to service the event. If a non-zero Repeat time is specified in the relevant line in the World File, it is a periodic event and must be returned to the event queue. A Repeat time of zero is mainly used by a TCP node to log packet arrivals. In this situation a separate LOG event prints out the details of each individual arrival, and is then discarded. Note: The only network entities that can be logged are nodes that implement the `log()` function, i.e. Transmit Processes cannot be logged, but the SLink they are associated with can.

A SERVICE event represents a specific task that a node must continually perform on itself, such as calculating average queue size (SFC Router) or checking for time-outs (TCP node). If defined, a node's `service()` function is called to

---

<sup>1</sup>Not to be confused with a packet's source and destination nodes.

perform the task. All SERVICE events are periodic and are therefore re-inserted into the event queue to perform the same task at a later time.

After each event is executed, the current simulation time is updated to reflect the time at which the event occurred. Once the current simulation time reaches the simulation duration time, processing of the event queue ends and the simulation terminates.

## 6 World File

A World File contains the specification of the topology of a particular network that will be simulated. This file allows the user to create, edit and delete a network from a single point, thus removing the need to manipulate source code files directly.

There are no restrictions on the name and suffix of World Files, but for the sake of convention, the suffix ".world" should be used. There is however, a restriction on the format of the content of a World File. Each World File must strictly adhere to a syntax to ensure that it is parsed into the simulator correctly.

**Refer to How to Create a World File (see Section 7 below) for instructions on creating a World File that follows the syntax.**

## 7 How to Create a World File

In the editor of your choice, create a new text file. The text file may be given any name, but the suffix .world must be used. The following steps must then be taken to correctly include all the entities that you wish to have in the network.

**Note: Follow the steps involving only the entities that are in your network. Throughout the process below, do not include the symbols < and > in the actual file. Comments can be included on a line that has two forward slashes “//” at the beginning.**

### 7.1 ID numbers

You must include a unique "id" number for each network object immediately after it's name, e.g. source greedy 21 ....

**Note: Id numbers must start from 1 and can only be positive integer values. The id numbers do not have to be sequential, but they must be specified in order. This must be done consistently throughout the world file.**

## 7.2 Define all Traffic Sources and Transmit Processes

### 7.2.1 Traffic Source

To define a Traffic Source (see Section 3.2 for information on Traffic Sources), the word "source" must precede the type of source, which can be either Greedy or Exponential. Their respective arguments must be included after the type.

The general structure for the definition of a Traffic Source object is as follows:

```
source <type> <id> <arg1> <arg2> ...
```

To define a Greedy Traffic Source, the following line must be present in the World File:

```
source greedy <id> <pac_size>
```

where:

*pac\_size* is the size of packets (in bytes) created by the Traffic Source.

To define an Exponential Traffic Source, the following line must be present:

```
source exponential <id> <pac_size> <mean_pac_sep>
```

where:

*pac\_size* is the same as for Greedy.

*mean\_pac\_sep* is the mean packet separation value which will be used to obtain an exponentially distributed value to separate packet departure times.

### 7.2.2 Transmit Process

To include a Transmit Process object (see Section 3.3 for more information on Transmit Processes) in the simulation, the definition in the World File must conform with the line:

```
process <type> <id> <arg1> <arg2> ...
```

To specify a Two\_state Transmit Process, the following line should be included:

```
process two_state <id> <good_rate> <bad_rate> <p>  
<q>
```

where:

*good\_rate* is the transmission rate in the "Good state" (in bits/sec).

*bad\_rate* is the transmission rate in the "Bad state" (in bits/sec).

*p* is the rate of change from good to bad state.

*q* is the rate of change from bad to good state.

To define a General Transmit Process, the following line should be included:

```
process general <id> <state_desc_file>
```

where:

*state\_desc\_file* is the name of the state description file. For more information regarding this file see the Reference Manual.

### 7.3 Define all nodes

Nodes are the tangible objects in a network and can be either of type Link, SLink, Generator, TCP, SFC\_Rec (SFC Filter), Router, or SFC\_Router.

#### 7.3.1 Links (Link, SLink)

A node of type Link or SLink (see Section 3.1 for more information on Links and SLinks) simulates the connection between two nodes.

To specify a standard link, the following line should be included:

```
node link <id> <rate> <prop_delay>
```

where:

*rate* is the transmission rate of the link (in bits/sec).

*prop\_delay* is the propagation delay of the link (in seconds).

To specify a Stochastic Link, the following line should be included:

```
node slink <id> <packet_xmit_func_id> <prop_delay>
```

where:

*packet\_xmit\_func\_id* is the id of the Transmit Process associated with the SLink.

*prop\_delay* is the same as for Link.

### 7.3.2 Flow Control Systems (Generator, TCP, SFC\_Rec)

There are also nodes which simulate flow control systems that can send and receive packets (Generator, TCP node can do both but a SFC Filter can only receive packets). Packets for transmission are created by the node's Traffic Source.

To specify a Generator (see Section 3.4 for more information on Generators), the following line should be included:

```
node generator <id> <node_to_connect_to> <window_size> <traffic_source_id>
```

where:

*node\_to\_connect\_to* is the id of the destination node of all packets transmitted by the Generator. If the Generator only receives packets, this argument should be made zero.

*window\_size* is the window size used for the transmission of packets. Also influences the amount of traffic in the network.

*traffic\_source\_id* is the id of the Traffic Source providing packets for transmission to the Generator. If the Generator only receives packets, this argument should be made zero.

To specify a TCP node (see Section 3.4 for more information on TCP nodes), the following line should be included:

```
node tcp <id> <dest_node> <max_seg_size> <traffic_source_id>
```

where:

*dest\_node* is the destination node for packets which are sent from the TCP node

*max\_seg\_size* is the maximum segment size of the packets (in bytes).

*traffic\_source\_id* is the same as for Generator.

To specify a node of type SFC\_Rec (SFC Filter) (see Section 3.4 for more information on SFC Filters), the following line should be included:

```
node sfc_filter <id> <max_delta_w> <tau>
```

where:

*max\_delta\_w* and

*tau* are parameters used to calculate window size in the SFC Filter.

### 7.3.3 Routers

The simulator has two types of routers which can be implemented. They are either a standard Router, or a Stochastic Flow Control Router (SFC\_Router).

To specify a standard Router (see Section 3.5 for more information on Routers), the following line should be included:

```
node router <id> <max_buffer_per_output_port>
```

where:

*max\_buffer\_per\_output\_port* is the size of the packet queue (buffer) for each output (port) of the Router (in bytes).

To specify a SFC\_Router (see Section 3.5 for more information on SFC Routers), the following line should be used:

```
node sfc_router <id> <max_buffer_per_output_port>  
<q_min> <b> <max_output_line_rate>
```

where:

*max\_buffer\_per\_output\_port* is the same as for Router.

*q\_min* and

*b* are parameters used to calculate the P(q) function of each packet. The P(q) function influences the flow control of a SFC Filter.

*max\_output\_line\_rate* is the maximum transmission rate of the outputs (in bits/sec).

## 7.4 Define all connections

Now that all the traffic sources, transmit processes and nodes have been defined, the connections between them must also be defined. Connections are strictly uni-directional, hence why separate connections must be defined for bi-directional traffic flow (e.g. define connection from node A to node B and also define connection from node B to node A).

To connect two nodes the following line should be included:

```
connect <source_node> <destination_node> [cost]
```

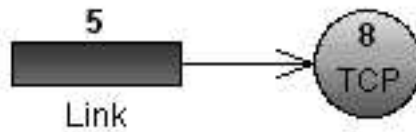
where:

*source\_node* is the id of the source node.

*destination\_node* is the id of the destination node.

*cost* is an optional field that represents the cost to get data from the source to the destination.

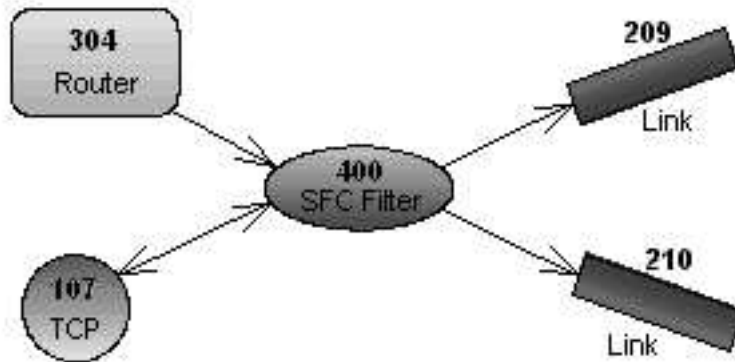
This process is straightforward for defining one to one connections from say a Link, which has one output, to a TCP node, which has one input. The configuration would look like this:



Traffic can only flow from Link 5 to TCP node 8, so the connection must be defined as:

```
connect 5 8
```

Connections involving Routers, SFC Routers and SFC Filters which can have multiple inputs and outputs are slightly more complicated. For the following example where a SFC Filter has two inputs and three outputs, each connection must be defined with the correct ordering of node ids. The configuration is depicted as follows:



It should be noted that one input of the Filter is connected to the output of the TCP node and one output of the Filter is connected to the input of the same TCP node. The connections are defined as:

```
connect 304 400  
connect 107 400  
connect 400 107
```

```
connect 400 209
connect 400 210
```

## 7.5 List the events

Now that the network has been created, events need to be scheduled to get the simulation started. There are two types of events which can be placed into the event queue prior to simulation. They are either one-off events that happen at a particular point in time or they are periodic events that occur at a set interval throughout the simulation. START and STOP events are strictly one-off events whereas LOG events can be either. E.g. a one-off LOG event for a TCP node triggers the automatic logging of packet arrivals as they occur, but periodic LOG events continuously occur at a set interval.

**Note: For all events in the World File, the argument <time> must not be less than zero.**

### 7.5.1 One-off events (START, STOP and LOG)

To begin operation of a node (see Section 5.2 for more information on START events), the following line must be included in the World File:

```
event <time> <node_id> start
```

where:

*time* is the time at which the node will be started.

*node\_id* is the id number of the node that will be started.

To terminate operation of a node (see Section 5.2 for more information on STOP events), the following line must be included:

```
event <time> <node_id> stop
```

where:

*time* is the time at which the node will stop operating.

*node\_id* is the id number of the node that will stop operating.

To create a non periodic LOG event (see Section 5.2 for more information on LOG events), the following line must be included in the World File:

```
event <time> <node_id> log 0
```

where:

*time* is the time that the LOG event will occur.

*node\_id* is the id of the node being logged.

### 7.5.2 Periodic Events (LOG)

To create a periodic event to regularly log various attributes of a node during simulation (see Section 5.2 for more information on LOG events), the following line must be used:

```
event <time> <node_id> log <repeat_time>
```

where:

*time* is the time at which the logging will commence. It must not be less than the start time of the node.

*node\_id* is the id of the node being logged.

*repeat\_time* is the interval between consecutive occurrences of the event, i.e. how often logging will occur.

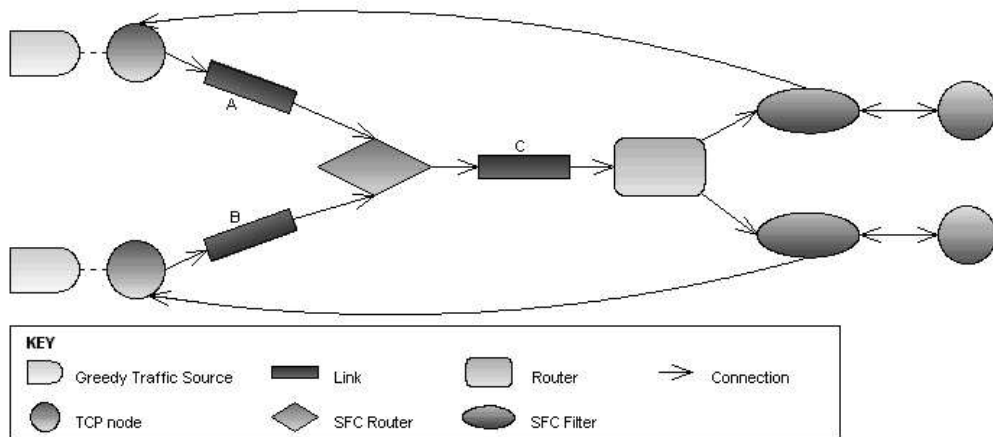
### 7.6 Specify the time at which simulation should end

The final step is to set the completion time of the simulation. This is done by having the word "end\_time" followed by a value, as follows:

```
end_time 1000
```

## 8 Construction of an Example World

An arbitrary network topology will be converted into a World File to illustrate the process described in the previous section. The network topology looks like:



This network consists of two Greedy Traffic Sources, two TCP transmitting nodes, two SFC Filters, two TCP receiving nodes, a Router, a SFC Router and three Links. The arrows indicate the direction of the traffic (connections) in the network. Throughout this process, all id numbers will be consecutive integer values starting from 1.

The first step in creating a World File is defining all the Traffic Sources and Transmit Processes. Since there are no Transmit Processes, only the Traffic Sources will be defined. The size of the packets created by the Traffic Sources will be 800 bytes:

```
// Greedy Traffic Sources
source greedy 1 800
source greedy 2 800
```

The next step is to define all the nodes. The TCP nodes have maximum segment sizes of 800 bytes and the two transmitters will send all their packets to the receiver nodes. The two TCP receiving nodes only absorb packets and therefore do not have their own Traffic Source:

```
// TCP transmitting nodes
node tcp 3 5 800 1
node tcp 4 6 800 2

// TCP receiving nodes
node tcp 5 0 800 0
node tcp 6 0 800 0
```

The SFC Filters have max\_delta\_w values of 4000 and 4000 and tau values of 2000 and 4000:

```
// SFC Filters
node sfc_filter 7 4000 2000
node sfc_filter 8 4000 4000
```

The Router has an output buffer size (buffer limit) of 32000 bytes:

```
// Router
node router 10 32000
```

The SFC Router has the same buffer limit as the Router, maximum output rate of 100Kbps (100,000 bits/sec) and values of 16,000 and 5 for q\_min and b respectively:

```
// SFC Router
node sfc_router 9 32000 16000 5 100000
```

Link A has transmission rate of 1Mbps (1 million bits/sec) and propagation delay of 0.5 seconds,

Link B has the same transmission rate as Link A and propagation delay of 0.1 seconds,

Link C has transmission rate of 100Kbps (100,000 bits/sec) and propagation delay of 0.05 seconds:

```
// Links
node link 11 1000000 0.5
node link 12 1000000 0.1
node link 13 100000 0.05
```

All the connections between the nodes must now be defined. The order in which they are written does not matter. There are 14 connections in total and they are:

```
// Connections
connect 3 11
connect 4 12
connect 11 9
connect 12 9
connect 9 13
connect 13 10
connect 10 7
connect 10 8
connect 7 5
connect 5 7
connect 8 6
connect 6 8
connect 7 3
connect 8 4
```

All that is remaining is to define any initial START, STOP and LOG events and enter the duration (end time) of the simulation. The only nodes that need to be explicitly started are the Generators/TCP nodes, and STOP events are not required for any of the nodes in the network. All nodes except the Traffic Sources and Links will be logged from the beginning. The initial events are:

```
// START events
event 0 3 start
event 0 4 start

// LOG events (all periodic with interval 0.1 seconds)
event 0 3 log 0.1
event 0 4 log 0.1
event 0 5 log 0.1
```

```
event 0 6 log 0.1
event 0 7 log 0.1
event 0 8 log 0.1
event 0.5 9 log 0.1
event 0 10 log 0.1
```

```
// End time
end_time 500
```

The process is complete! We now have a textual representation of the network that we can simulate with the QNS. The complete World File can be seen in the subsection below.

## 8.1 Complete World File

```
// Greedy Traffic Sources
source greedy 1 800
source greedy 2 800

// TCP transmitting nodes
node tcp 3 5 800 1
node tcp 4 6 800 2

// TCP receiving nodes
node tcp 5 0 800 0
node tcp 6 0 800 0

// SFC Filters
node sfc_filter 7 4000 2000
node sfc_filter 8 4000 4000

// Router
node router 10 32000

// SFC Router
node sfc_router 9 32000 16000 5 100000

// Links
node link 11 1000000 0.5
node link 12 1000000 0.1
node link 13 100000 0.05

// Connections
connect 3 11
connect 4 12
connect 11 9
```

```

connect 12 9
connect 9 13
connect 13 10
connect 10 7
connect 10 8
connect 7 5
connect 5 7
connect 8 6
connect 6 8
connect 7 3
connect 8 4

// START events
event 0 3 start
event 0 4 start

// LOG events
event 0 3 log 0.1
event 0 4 log 0.1
event 0 5 log 0.1
event 0 6 log 0.1
event 0 7 log 0.1
event 0 8 log 0.1
event 0.5 9 log 0.1
event 0 10 log 0.1

// End time
end_time 500

```

## 9 How to Add a New Module

To add newly created module into the QNS, minor modifications must be made to a small set of existing source files. The source code of the module itself should be written by the user and must conform to various templates included in this manual. The syntax of the World File must be altered to specify how the new module can be included in network topologies. Finally, the Makefile requires modification to ensure the correct compilation of all sources files.

The subsections below outline the content requirements of the source files that the user must be aware of (and adhere to) when developing new modules.

The sections after these will outline the steps involved in creating and incorporating new modules into the QNS. Each section describes the process for a module that extends either of the following entities: Router, Traffic Source, Transmit Process, Link, Generator.

## 9.1 The Constructor

Each network object in the QNS is referenced and manipulated by referring to its own unique identification (id) number (see Section 7.1), so the constructor of every module's class definition must have "const int id" as its first argument. The id numbers of all objects are assigned by the user in the World File.

All other inputs for the constructor will be specific parameters (predominantly user-defined) that are necessary for the node to achieve its goals. For example, a link which connects two nodes that can send and/or receive packets of data will need the user-defined rate and propagation delay as inputs to its constructor.

Constructors of all modules (except Transmit Processes) must initialise an instance of their parent class as well as their own, with the same id number and any other necessary arguments that need to be set. E.g. the constructor of a TCP node initialises an instance of a Generator class in the following way:

```
TCP::TCP(const int id, const int d, const int mss, Traffic_source
*s):Generator::Generator(id, d, 1, s)
```

This method is called using an "initialisation list". The Generator object is given the same values of the TCP node's arguments id, d and s. The Generator's constructor would also initialise a Node object with the same id number. This creates the situation where within the TCP node object there is a Generator object and within the Generator object there is a Node object (the Node object is initialised in the Generator class). This ensures that the event handler, Node\_map and Con\_table can manipulate each node through its instance of the Node class.

## 9.2 Required Functions for Routers, Generators and Links

Apart from the constructor, all other functions that need to be implemented must directly (or indirectly) service some or all of the events: START, STOP, PASS, SERVICE, LOG. For information on the arguments required to schedule events, refer to the Event class in the Reference Manual.

### 9.2.1 Functions to service event START or STOP

For an event of type START and STOP (see Section 5.2), the functions start() and stop() are defined in the base class "Node". These functions determine whether a node can send and/or receive packets. If the node must perform other routines when it starts or stops, the functions can be redefined in the node's source code.

### 9.2.2 Function to service event PASS

An event of type PASS (see Section 5.2) represents the process of a packet being transmitted from one node (the source) to another (the immediate destination)

in the network. The functions that must be implemented in an event of type PASS are `out()` for the source node and `in()` for the destination node. Nodes that have more than one output (e.g. a router) require that the event contains extra information to indicate which output will send out the packet. The `out_event()` function takes an event as input and can be implemented for this purpose instead of `out()`.

### 9.2.3 Function to service event SERVICE

The SERVICE event (see Section 5.2) is the mechanism that allows a node to perform a recurring task on itself at a constant interval. The interval is set by the user and passed as a parameter to the SERVICE event when created and scheduled. Because this type of event is periodic, the event handler re-inserts the event object back into the event queue after servicing it. If the periodic task is required by the node, it must be defined in the node's `service()` function, which is otherwise left empty. In the TCP node's class the service function checks for time-outs, in the SFC Router's class, it calculates the average queue size.

### 9.2.4 Function to service event LOG

Nearly all nodes have an implementation of the function `log()` to service LOG events (see Section 5.2). `Log()` prints out the type of node, its id, the current time, then some useful statistics which are directly related to that particular type of node. Any new node that is able to print useful statistics to standard output, should include a definition of this function. A node that is not required to do this must have an empty function defined.

All other functions are technically not required, unless:

- they perform some other tasks not specified above,
- they modify the members of the node,
- they assist in servicing the events START, STOP, PASS, SERVICE, LOG.

## 9.3 Required Functions for Traffic Sources

Traffic Sources are different to Routers, Links and Generators in that they do not require explicit implementations for the same functions that the other nodes service events with. Their only task is to output packets to Generators, therefore the only function that requires a definition is `out()` (or `out_event()`). This definition is provided by the abstract class `Traffic_source`. `Traffic_source` also provides empty definitions for the functions `in()` and `service()`. If the user believes that a logging function is beneficial (e.g. to indicate each packet's creation), a definition of `log()` should also be included.

The declaration of the virtual function `get()` in `Traffic_source` requires that the user implement a function of the same name in any new Traffic Source. This

function basically determines the method or rate at which the Traffic Source outputs packets to a Generator and must produce PASS (see Section 5.2) events to indicate when packets are ready to be forwarded.

See the QNS Reference Manual for more information.

## 9.4 Required Functions for Transmit Processes

Transmit Processes are not derived from node objects. Their purpose is to model packet transmission over a stochastic link (SLink). Like the other network entities, Transmit Processes must be assigned unique id numbers. The objects themselves however, are manipulated entirely within SLink objects, therefore they do not have to record their own id numbers. The abstract class `Transmit_process` declares three functions as purely virtual, i.e. they require implementation in every Transmit Process class that is defined.

The function `transmit()` simulates the actual transmission of a packet down the slink. The function `time()` should calculate the transmission time of the packet and the function `lost()` must determine whether packet loss occurs during transmission.

See the QNS Reference Manual for more information.

## 10 Creating a Module of Type Router

The following steps must be completed to successfully create and use a new type of Router in the QNS. Throughout the section, the name "New Router" will be used for the module. This can be substituted with an arbitrary name as chosen by the user. Specific examples of functions and other members are provided (in the templates for the source files) below the lines beginning with: `// Specific example`. Also, brackets of the kind `<` and `>` should not be included in the actual source code, as they are merely place holders in the fragments of code below.

### 10.1 Create a .h file

The header file must be written to enable other modules to communicate with the new Router and vice versa. The class definition of the new Router and any other relevant data members or structures should be defined in the header file.

The following is a template illustrating the required contents of the header file `New_Router.h`:

```
// New_Router.h

// Must include these lines
#ifndef NEW_ROUTER_H
#define NEW_ROUTER_H
```

```

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.

#define:  - Any necessary constants

// Specific example:
// This value is the interval at which the service function will be
// periodically queued in the event list.
#define NEW_ROUTER_SERVICE_INTERVAL 0.01

// Definition of any necessary structures.
.
.
.

// Class definition
class New_Router : public Router {

// Declaration of necessary members, or instantiations of structures
// defined above.
.
.
.

// Public functions
public:
// Specific example:
// The constructor and destructor.
New_Router(unsigned int id, any other required parameters );
~New_Router(void){}

// Specific examples:
// An implementation of out_event() must be included in the .cpp
// file to obtain the output port number that the packet will
// be sent from.
Packet* out_event(Event *the_event);

// The log() function must be defined in the .cpp file to output
// specific information related to the New Router.
void log(void);

// If the New Router needs to perform periodic checks, or updating
// of variables, then it will need to have it's own definition
// for service() in the .cpp file.

```

```

        void service(void);

// Private functions
private:
    // Any functions that are internal to this object and assist
    // in servicing the various types of events described above should
    // be defined as private.

// Protected functions
protected:
    // If there is scope for deriving this class, it would be wise
    // to define some functions as protected, to enable subclasses
    // to have access to them.

};

// Must include this line
#endif

```

## 10.2 Create a .cpp file

The source file must be written to provide implementations for the functions listed in the class definition in `New_Router.h` (that haven't already been defined there).

Note: the header file `New_Router.h` must be `#included` at the top of the `.cpp` file.

The following is a template of what `New_Router.cpp` should look like:

```

// New_Router.cpp

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.
          New_Router.h

// New_Router may need none, some, or all of the following global variables
// which are used in the simulations.

extern Event_list event_list;
extern Node_map node_map;
extern Con_table con_table;

```

```

// Definition of the constructor

// Specific example:
// Definition of the constructor that will initialise the instance
// of Router with the same id and a predetermined buffer length bl.
New_Router::New_Router(unsigned int id, all other parameters needed):Router(id,
bl) {

    // Initialise internal variables and attributes.

    // Schedule any events of type SERVICE either here or in a definition
    // of start().

// Specific example:
// Add the service event - a repeating event which will periodically
// update all estimators etc.
Event *e = new Event(0, id(), SERVICE, New_Router_SERVICE_INTERVAL);
event_list.insert(e);

}

void New_Router::in(Packet *p)
{
    // This function must process an incoming packet. The basic Router
    // performs the following tasks:
    // * checks packet's TTL (Time To Live) field.
    // * determines the output port that packet will be placed on.
    // * records any packet losses which occur due to congestion at
    // output.
    // * schedules an event to output the packet.

}

Packet* New_Router::out_event(Event *the_event)
{
    // This function can perform the following steps, or call another
    // function to perform them:
    // * obtains the required data from the_event that specifies
    // which output port the packet will leave from.
    // * removes the packet from the appropriate queue.
    // * updates any state variables.
    // * returns the packet.
}

```

```

}

void New_Router::log()
{
    // The following is an example of an implementation of a log function.
    // Most log functions will preferably follow this basic format.

    // Specific example:
    ostream m;

    m << // All the relevant variables in a clear and simple format.
    m << // ....
    .
    .

    // The following example is from SFC_router:
    /*
        m << " queue: " << Router::mem_used() << " Bytes";
        m << " Packets dropped: " << Router::packets_dropped();
        m << " ave output rate: " << SFC_chan_rate;
        m << " cur p(q): " << calc_pq();
    */

    cout << "New Router Node" << id() << "_ " << '<' <<
    event_list.get_cur_time() << '>';
    cout << " :";
    cout << m.str() << endl;
}

void New_Router::service(void)
{
    // Complete any routine calculations, checks, or updates which are
    // necessary.
}

```

Any other functions which are defined will be used to assist in the implementation of the functions defined in the template.

### 10.3 Modify main.h

The number of input variables used by the new Router must be stipulated by adding the following line of code to main.h (anywhere between the lines: #de-

fine MAIN\_H and #endif):

```
#define NEW_ROUTER_ARGS <number of arguments to the constructor>
```

If the number of arguments exceeds the absolute maximum number of arguments (ten), the line:

```
#define MAX_ARGS 10
```

can be altered accordingly.

## 10.4 Modify main.cpp

This file must be modified to ensure that the New Router can be used in simulations.

The first addition to the file is the inclusion of the header file New\_Router.h with the following line:

```
#include "New_Router.h"
```

This line should be placed anywhere among the lines where other .h files are #included.

The second addition to the file is the inclusion of source code in the function build\_node(). The following code must be added before the last else-statement:

```
else if(word == "new_router")
// The string "new_router" is found in world files that use the
// New Router.
{
    // This obtains all the input arguments for the module from
    // the world file.
    for(i=0; i<NEW_ROUTER_ARGS;i++) args[i] = get_next_word(line,
&pos);

    // Create a New_Router object.
    // Note: atof should be used if any argument is a floating
    // point value.
    New_Router *new_router = new New_Router(atof(args[0].c_str()),
atof(args[1].c_str()), ..., atof(args[N].c_str()));

    // Insert a pointer to the object into the node_map.
    node_map.add_node((Node*)router, atof(args[0].c_str()));

    // Confirmation that the New Router was created.
```

```

    cerr << "Added New Router:  " << atoi(args[0].c_str()) << endl;
}

```

## 10.5 Modify the Makefile

One minor alteration must be made to the Makefile to ensure that the source code of the New Router compiles correctly.

On the line beginning with the words "CORE\_OBJ", there is a list of filenames with a .o suffix. The filename of the New Router with a .o extension added to it must be inserted at the end of the list as shown below:

```

CORE_OBJ = main.o routes.o ... tp_general.o New_Router.o

```

## 10.6 Note the changes to the World File syntax

The final step of the process involves adding a new line to the syntax of the world file. This does not need to be done explicitly, but it would demonstrate to any subsequent users how to define the New Router in a world file.

A line similar to the following one should be included in the section where all other Routers are defined:

```

node new_router <id> <arg1> <arg2> ... <argN>

```

where:

**node:** specifies that the New Router is an object of type node.

**new\_router:** is the string used to identify the New Router in the world file.

**id:** is the unique id number of the New Router.

**<arg1> <arg2> ... <argN>:** are user defined arguments required for the implementation of the New Router. These are the inputs to the New Router's constructor.

## 11 Creating a Module of Type Generator

The following tasks must be completed to successfully create and utilise new types of Generators in the QNS. Throughout the section, the name "New Generator" will be used to name the module and it can be replaced with an arbitrary name of the user's choice. Specific examples of functions and other members are provided (in the templates for the source files) below the lines beginning with: // Specific example.

Also, brackets of the kind < and > should not be included in the actual source code, as they are merely place holders in the fragments of code below.

## 11.1 Create a .h file

A header file must be written to enable other modules to communicate with the New Generator and vice versa. The class definition of the New Generator and any other relevant data members or structures should be defined in the header file.

The following is a template illustrating the required contents of the header file `New_Generator.h`:

```
// New_Generator.h

// Must include these lines
#ifndef NEW_GENERATOR_H
#define NEW_GENERATOR_H

#include: - standard library files.
         - header files from the simulator.
         - any other headers.

#define: - Any necessary constants

// Specific example:
// This value is the interval at which the service function for New
// Generator will be periodically queued in the event list.
#define NEW_GENERATOR_SERVICE_INTERVAL 0.01

// Definition of any necessary structures, e.g. structures that keep
// track of state variables, such as tcp_opt, or that describe payloads
// of packets that are generated by the New Generator.
.
.
.

// Class definition
class New_Generator : public Generator {

// Declaration of necessary members, or instantiations of structures
// defined above.
.
.
.

// Public functions
public:
// Specific example:
```

```

// The constructor and destructor.
New_Generator(unsigned int id, other parameters, Traffic_source
*s);
~New_Generator(void){}

// Specific examples:
// An implementation of out() must be included in the .cpp file
// to determine the tasks that will be performed when a packet
// is output from the New Generator.
Packet* out(void);

// The log() function must be defined in the .cpp file to output
// specific information related to the New Generator, such as
// packet arrival time statistics.
void log(void);

// If the New Generator needs to perform periodic checks, or
// updating of variables, then it will need to have it's own
// definition for service() in the .cpp file.
void service(void);

// When any Generator comes online for the first time, a certain
// task must be performed. E.g. a regular Generator inserts
// a number of initial packets into the system, a TCP node initiates
// the service event that checks for time-outs and also inserts
// a packet into the system. All of these are defined in their
// respective start() functions.
void start(void);

// Private functions
private:
// Any functions that are internal to this object and assist
// in servicing the various types of events described above should
// be defined as private.

// Protected functions
protected:
// If there is scope for deriving this class, it would be wise
// to define some functions as protected, to enable subclasses
// to have access to them.

```

```
};

// Must include this line
#endif
```

## 11.2 Create a .cpp file

The source file must be written to provide implementations for the functions listed in the class definition in `New_Generator.h` (that haven't already been defined there).

Note: the header file `New_Generator.h` must be `#included` at the top of the `.cpp` file.

The following is a template of what `New_Generator.cpp` should look like:

```
// New_Generator.cpp

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.
          New_Generator.h

// New_Generator may need none, some, or all of the following global
// variables which are used in the simulations.

extern Event_list event_list;
extern Node_map node_map;
extern Con_table con_table;

// Definition of the constructor

// Specific example:
// Definition of the constructor that will initialise the instance
// of Generator with the same id and predetermined variables d and
// w. Since each Generator can emit packets, a Traffic Source (s)
// must be included as the last of the input parameters to the constructor.
New_Generator::New_Generator(unsigned int id, other parameters, Traffic_source
*s):Generator(id, d, w, s) {

    // Initialise internal variables and attributes.

    // Schedule any events of type SERVICE either here or in a definition
    // of start().

// Specific example:
```

```

    // Add the service event - a repeating event which will periodically
    // update all estimators etc.
    Event *e = new Event(0, id(), SERVICE, NEW_GENERATOR_SERVICE_INTERVAL);
    event_list.insert(e);
}

void New_Generator::in(Packet *p)
{
    // This function must process an incoming packet in a certain way,
    // depending upon the value of the packet's protocol field, and
    // then schedule an event to output the packet if it hasn't reached
    // it's destination yet.
}

Packet* New_Generator::out(void)
{
    // This function must remove the next packet from the New Generator's
    // output buffer and perform any necessary tasks, such as putting
    // a time-stamp on the packet, as well.
}

void New_Generator::log()
{
    // The following is an example of an implementation of a log function.
    // Most log functions will preferably follow this basic format.

    // Specific example:
    ostringstream m;

    m << // All the relevant variables in a clear and simple format.
    m << // ....
    .
    .

    // The following example is from TCP:
    /*
        m << " snd_next: " << tp.snd_next;
        m << " rcv_next: " << tp.rcv_next;
        m << " last_ack: " << tp.last_ack;
    */
}

```

```

        m << " cwnd: " << tp.snd_cwnd*tp.mss;
        m << " cwnd_clamp: " << tp.snd_cwnd_clamp*tp.mss;
        m << " awnd: " << tp.rcv_wnd*tp.mss;
        m << " rto: " << tp.rto;
        m << " last_rtt: " << tp.last_rtt;
        m << " ss_thresh: " << tp.snd_ssthresh;
    */

    cout << "New Generator Node" << id() << "_ " << '<' <<
    event_list.get_cur_time() << '>';
    cout << " :";
    cout << m.str() << endl;

}

void New_Generator::service(void)
{
    // Complete any routine calculations, checks, or updates which are
    // necessary.
}

```

Any other functions which are defined will be used to assist in the implementation of the functions defined in the template.

### 11.3 Modify main.h

The number of input variables used by the New Generator must be specified by adding the following line of code to main.h (anywhere between the lines #define MAIN\_H and #endif):

```
#define NEW_GENERATOR_ARGS <number of arguments to the constructor>
```

If the number of arguments exceeds the absolute maximum number of arguments (ten), the line:

```
#define MAX_ARGS 10
```

can be altered accordingly.

### 11.4 Modify main.cpp

This file must be modified to ensure that the New Generator can be used in simulations.

The first addition to the file is the inclusion of the header file `New_Generator.h` with the following line:

```
#include "New_Generator.h"
```

This line should be placed anywhere among the lines where other `.h` files are `#included`.

The second addition to the file is the inclusion of source code in the function `build_node()`. The following code must be added before the last `else`-statement:

```
else if(word == "new_generator")
// The string "new_generator" is found in world files that use the
// New Generator.
{
    // This obtains all the input arguments for the module from the
    // world file.
    for(i=0; i<NEW_GENERATOR_ARGS;i++)
        args[i] = get_next_word(line, &pos);

    // The following code sets up a Traffic Source that the New Generator
    // can be attached to.
    Traffic_source *ts;

    if(atoi(args[NEW_GENERATOR_ARGS-1].c_str()) == 0)
        ts = NULL;
    else
        ts = (Traffic_source*)node_map.id2node(atoi(args[NEW_GENERATOR_ARGS-1].
        c_str()));

    // Create a New_Generator object.
    // Note: atof should be used if argument is a floating point value.
    New_Generator *new_generator = new New_Generator(atoi(args[0].c_str()),
    atoi(args[1].c_str()), ..., atoi(args[N].c_str()), ts);

    // Insert a pointer to the object into the node_map.
    node_map.add_node((Node*)new_generator, atoi(args[0].c_str()));

    // Confirmation that the New Generator was created.
    cerr << "Added New Generator: " << atoi(args[0].c_str()) << endl;
}
```

## 11.5 Modify the Makefile

One minor alteration must be made to the Makefile to ensure that the source code of the New Generator compiles correctly.

On the line beginning with the words "CORE\_OBJ", there is a list of filenames with a .o suffix. The filename of the New Generator with a .o extension added to it must be inserted at the end of the list as shown below:

```
CORE_OBJ = main.o routes.o ... tp_general.o New_Generator.o
```

## 11.6 Note the changes to the World File syntax

The final step of the process involves adding a new line to the syntax of the world file. This does not need to be done explicitly, but it would demonstrate to any subsequent users how to define the New Generator in a world file.

A line similar to the following one should be included in the section where all other Generators are defined:

```
node new_generator <id> <arg1> <arg2> ... <argN>
```

where:

**node:** specifies that the New Generator is an object of type node.

**new\_generator:** is the string used to identify the New Generator in the world file.

**id:** is the unique id number of the New Generator.

**<arg1> <arg2> ... <argN>:** are user defined arguments required for the implementation of the New Generator. These are the inputs to the New Generator's constructor.

## 12 Creating a Module of Type Link

The following tasks must be completed to successfully create and utilise new types of Links in the QNS. Throughout the section, the name "New Link" will be used to name the module and it can be substituted with an arbitrary name of the user's choice. Specific examples of functions and other members are provided, in the templates for the source files, after the line: // Specific example. Also, brackets of the kind < and > should not be included in the actual source code, as they are merely place holders in the fragments of code below.

## 12.1 Create a .h file

A header file must be written to enable other modules to communicate with the New Link and vice versa. The class definition of the New Link and any other relevant data members or structures should be defined in the header file.

The following is a template illustrating the required contents of the header file `New_Link.h`:

```
// New_Link.h

// Must include these lines
#ifndef NEW_LINK_H
#define NEW_LINK_H

#include: - standard library files.
         - header files from the simulator.
         - any other headers.

#define: - Any necessary constants

// Definition of any necessary structures
.
.
.

// Class definition
class New_Link : public Link {

// Declaration of necessary members, or instantiations of structures
// defined above.
.
.
.

// Public functions
public:
// Specific example:
    // The constructor and destructor.
    New_Link(unsigned int id, other parameters, Traffic_source *s);
    ~New_Link(void){}

// Specific examples:
    // An implementation of out() must be included in the .cpp file
    // to determine the tasks that will be performed when a packet
    // is output from the New Link.
```

```

    Packet* out(void);

    // The log() function must be defined in the .cpp file to output
    // specific information related to the New Link.
    void log(void);

    // If the New Link needs to perform periodic checks, or
    // updating of variables, then it will need to have it's own
    // definition for service() in the .cpp file. Usually not implemented
    // for links.
    void service(void);

// Private functions
private:
    // Any functions that are internal to this object and assist
    // in servicing the various types of events described above should
    // be defined as private.

// Protected functions
protected:
    // If there is scope for deriving this class, it would be wise
    // to define some functions as protected, to enable subclasses
    // to have access to them.

};

// Must include this line
#endif

```

## 12.2 Create a .cpp file

The source file must be written to provide implementations for the functions listed in the class definition in `New_Link.h` (that haven't already been defined there).

Note: the header file `New_Link.h` must be `#included` at the top of the `.cpp` file.

The following is a template of what `New_Link.cpp` should look like:

```

// New_Link.cpp

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.
          New_Link.h

```

```

// New_Link may need none, some, or all of the following global
// variables which are used in the simulations.

extern Event_list event_list;
extern Node_map node_map;
extern Con_table con_table;

// Definition of the constructor

// Specific example:
// Definition of the constructor that will initialise the instance
// of Link with the same id and values of transmission rate and propagation
// delay.
New_Link::New_Link(unsigned int id, any other required parameters):Link(id,
ra, del) {

    // Initialise internal variables and attributes.

    // Schedule any events of type SERVICE either here or in a definition
    // of start(). Usually not implemented by links.

}

void New_Link::in(Packet *p)
{
    // This function must process an incoming packet. A standard Link
    // calculates the transmission time and delay of the packet that
    // will travel down the link and schedules an event to output it
    // when it has reached the other end.

}

Packet* New_Link::out(void)
{
    // This function must remove the next packet from the New Link's
    // output buffer and perform any necessary tasks.

}

void New_Link::log()

```

```

{
    // The following is an example of an implementation of a log function.
    // Most log functions will preferably follow this basic format.

// Specific example:
    ostream m;

    m << // All the relevant variables in a clear and simple format.
    m << // ....
    .
    .

// The following example is from SLink:
/*
    m << " in_flight: " << in_flight() << " xmit_time: " ;
    m << cur_xmit_time;
*/

    cout << "New Link Node" << id() << "_ " << '<' <<
    event_list.get_cur_time() << '>';
    cout << " :";
    cout << m.str() << endl;
}

void New_Link::service(void)
{
    // Complete any routine calculations, checks, or updates which are
    // necessary. Usually left empty for a link.
}

```

Any other functions which are defined will be used to assist in the implementation of the functions defined in the template.

### 12.3 Modify main.h

The number of input variables used by the New Link must be specified by adding the following line of code to main.h (anywhere between the lines #define MAIN\_H and #endif):

```
#define NEW_LINK_ARGS <number of arguments to the constructor>
```

If the number of arguments exceeds the absolute maximum number of arguments (ten), the line:

```
#define MAX_ARGS 10
```

can be altered accordingly.

## 12.4 Modify main.cpp

This file must be modified to allow the New Link to be used in simulations.

The first addition to the file is the inclusion of the header file `New_Link.h` with the following line:

```
#include "New_Link.h"
```

The second addition to the file is the inclusion of source code in the function `build_node()`. The following code must be added before the last else-statement:

```
else if(word == "new_link")
// The string "new_link" is found in world files that use the
// New Link.
{
    // This obtains all the input arguments for the module from the
    // world file.
    for(i=0; i<NEW_LINK_ARGS;i++)
        args[i] = get_next_word(line, &pos);

    // Create a New_Link object.
    // If a Transmit Process is required as an argument (similar to
    // SLink), the last argument in the following line should be
    // included.
    // Note: atof should be used if argument is a floating point value.
    New_Link *new_link = new New_Link(atoi(args[0].c_str()),
    atoi(args[1].c_str()), ..., atoi(args[N].c_str()),
    trans_procs[atoi(args[1].c_str())]);

    // Insert a pointer to the object into the node_map.
    node_map.add_node((Node*)new_link, atoi(args[0].c_str()));

    // Confirmation that the New Link was created.
    cerr << "Added New Link: " << atoi(args[0].c_str()) << endl;
}
```

## 12.5 Modify the Makefile

One minor alteration must be made to the Makefile to ensure that the source code of the new Link compiles correctly.

On the line beginning with the words "CORE\_OBJ", there is a list of filenames with a .o suffix. The filename of the New Link with a .o extension added to it must be inserted at the end of the list as shown below:

```
CORE_OBJ = main.o routes.o ... tp_general.o New_Link.o
```

## 12.6 Note the changes to the World File syntax

The final step of the process involves adding a new line to the syntax of the world file. This does not need to be done explicitly, but it would demonstrate to any subsequent users how to define the New Link in a world file.

A line similar to the following one should be included in the section where all other Links are defined:

```
node new_link <id> <arg1> <arg2> ... <argN>
```

where:

*node*: specifies that the New Link is an object of type node.

*new\_link*: is the string used to identify the New Link in the world file.

*id*: is the unique id number of the New Link.

*<arg1> <arg2> ... <argN>*: are user defined arguments required for the implementation of the New Link. These are the inputs to the New Link's constructor.

# 13 Creating a Module of Type Traffic Source

The following tasks must be completed to successfully create and utilise new types of Traffic Sources in the QNS. Throughout the section, the name "New Traffic Source" will be used to name the module and it can be substituted with an arbitrary name of the user's choice. Specific examples of functions and other members are provided, in the templates for the source files, after the line: // Specific example. Also, brackets of the kind < and > should not be included in the actual source code, as they are merely place holders in the fragments of code below.

## 13.1 Create a .h file

A header file must be written to enable other modules to communicate with the New Traffic Source and vice versa. The class definition of the New Traffic

Source and any other relevant data members or structures should be defined in the header file.

The following is a template illustrating the required contents of the header file `New_Traffic_Source.h`:

```
// New_Traffic_Source.h

// Must include these lines
#ifndef NEW_TRAFFIC_SOURCE_H
#define NEW_TRAFFIC_SOURCE_H

#include: - standard library files.
         - header files from the simulator.
         - any other headers.

#define: - Any necessary constants

    // Definition of any necessary structures.
    .
    .
    .

// Class definition
class New_Traffic_Source : public Traffic_source {

// Declaration of necessary members, or instantiations of structures
// defined above.
    .
    .
    .

// Public functions
public:
// Specific example:
    // The constructor and destructor.
    New_Traffic_Source(unsigned int id, any other required parameters);
    ~New_Traffic_Source(void){}

// Specific examples:
    // An implementation of get() must be included in the .cpp file
    // to determine how and at what interval (if any) the New Traffic
    // Source will create packets.
    Packet* get(const int dest);

    // The log() function must be defined in the .cpp file to output
```

```

        // specific information related to the New Traffic Source.
        void log(void);

// Private functions
private:
    // Any functions that are internal to this object and assist
    // in servicing the various types of events described above should
    // be defined as private.

// Protected functions
protected:
    // If there is scope for deriving this class, it would be wise
    // to define some functions as protected, to enable subclasses
    // to have access to them.

};

// Must include this line
#endif

```

## 13.2 Create a .cpp file

The source file must be written to provide implementations for the functions listed in the class definition in `New_Traffic_Source.h` (that haven't already been defined there).

Note: the header file `New_Traffic_Source.h` must be `#included` at the top of the `.cpp` file.

The following is a template of what `New_Traffic_Source.cpp` should look like:

```

// New_Traffic_Source.cpp

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.
          New_Traffic_Source.h

// New_Traffic_Source may need none, some, or all of the following
global
// variables which are used in the simulations.

extern Event_list event_list;
extern Node_map node_map;

```

```

extern Con_table con_table;

// Definition of the constructor

// Specific example:
// Definition of the constructor that will initialise the instance
// of Traffic Source with the same id number.
New_Traffic_Source::New_Traffic_Source(unsigned int id, any other required
parameters):Traffic_source(id) {

    // Initialise internal variables and attributes.

}

void New_Traffic_Source::get(const int dest)
{
    // This function must create a new packet and initialise its
    // IPHeader fields (the protocol value must be set to RAW) and
    // payload. The packet is then queued on the output buffer and
    // an event is scheduled to output the packet at a set time in the
    // future. Ts_greedy outputs packets as they are created, ts_exp
    // makes them available after an exponential time.

}

void New_Traffic_Source::log()
{
    // The following is an example of an implementation of a log function.
    // Most log functions will preferably follow this basic format.

// Specific example:
    ostream m;

    m << // All the relevant variables in a clear and simple format.
    m << // ....
    .
    .

    cout << "New Traffic Source Node" << id() << "_ " << '<' <<
    event_list.get_cur_time() << '>';
    cout << " :";
    cout << m.str() << endl;
}

```

```
}
```

Any other functions which are defined will be used to assist in the implementation of the functions defined in the template.

### 13.3 Modify main.h

The number of input variables used by the New Traffic Source must be specified by adding the following line of code to main.h (anywhere between the lines `#define MAIN_H` and `#endif`):

```
#define NEW_TRAFFIC_SOURCE_ARGS <number of arguments to the constructor>
```

If the number of arguments exceeds the absolute maximum number of arguments (ten), the line:

```
#define MAX_ARGS 10
```

can be altered accordingly.

### 13.4 Modify main.cpp

This file must be modified to allow the New Traffic Source to be used in simulations.

The first addition to the file is the inclusion of the header file `New_Traffic_Source.h` with the following line:

```
#include "New_Traffic_Source.h"
```

This line should be placed anywhere among the lines where other `.h` files are `#included`.

The second addition to the file is the inclusion of source code in the function `add_source()`. The following code must be added before the last `else`-statement:

```
else if(word == "new_traffic_source")
// The string "new_traffic_source" is found in world files that use
// the New Traffic Source.
{
// This obtains all the input arguments for the module from
// the world file.
for(i=0; i<NEW_TRAFFIC_SOURCE_ARGS;i++)
    args[i] = get_next_word(line, &pos);
```

```

// Create a New_Traffic_Source object.
// Note: atof should be used if argument is a floating point value.
New_Traffic_Source *ts = new New_Traffic_Source(atof(args[0].c_str()),
atoi(args[1].c_str()), ..., atoi(args[N].c_str()));

// Insert a pointer to the object into the node_map.
node_map.add_node((Node*)ts, atoi(args[0].c_str()));

// Confirmation that the New Traffic Source was created.
cerr << "Added New Traffic Source: " << atoi(args[0].c_str()) <<
endl;
}

```

### 13.5 Modify the Makefile

One minor alteration must be made to the Makefile to ensure that the source code of the New Traffic Source compiles correctly.

On the line beginning with the words "CORE\_OBJ", there is a list of filenames with a .o suffix. The filename of the New Traffic Source with a .o extension added to it must be inserted at the end of the list as shown below:

```
CORE_OBJ = main.o routes.o ... tp_general.o New_Traffic_Source.o
```

### 13.6 Note the changes to the World File syntax

The final step of the process involves adding a new line to the syntax of the world file. This does not need to be done explicitly, but it would demonstrate to any subsequent users how to define the New Traffic Source in a world file. All Traffic Sources must be defined in world files before they are assigned to Generators.

A line similar to the following one should be included in the section where all other Traffic Sources are defined:

```
source new_traffic_source <id> <arg1> <arg2> ... <argN>
```

where:

**source:** specifies that the New Traffic Source is a Traffic Source object.

**new\_traffic\_source:** is the string used to identify the New Traffic Source in the world file.

**id:** is the unique id number of the New Traffic Source.

**<arg1> <arg2> ... <argN>:** are user defined arguments required for the implementation of the New Traffic Source. These are the inputs to the New Traffic Source's constructor.

## 14 Creating a Module of Type Transmit Process

The following tasks must be completed to successfully create and utilise new types of Transmit Processes in the QNS. Throughout the section, the name "New Transmit Process" will be used to name the module and it can be replaced with an arbitrary name of the user's choice. Specific examples of functions and other members are provided (in the templates for the source files) below the lines beginning with: // Specific example. Also, brackets of the kind < and > should not be included in the actual source code, as they are merely place holders in the fragments of code below.

### 14.1 Create a .h file

A header file must be written to enable other modules to communicate with the New Transmit Process and vice versa. The class definition of the New Transmit Process and any other relevant data members or structures should be defined in the header file.

The following is a template illustrating the required contents of the header file New\_Transmit\_Process.h:

```
// New_Transmit_Process.h

// Must include these lines
#ifndef NEW_TRANSMIT_PROCESS_H
#define NEW_TRANSMIT_PROCESS_H

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.

#define:  - Any necessary constants

    // Definition of any necessary structures.
    .
    .
    .

// Class definition
class New_Transmit_Process : public Transmit_process {

// Declaration of necessary members, or instantiations of structures
// defined above.
    .
    .
    .
```

```

// Public functions
public:
// Specific example:
    // The constructor and destructor.
    New_Transmit_Process(any required user-defined parameters);
    ~New_Transmit_Process(void){}

// Specific examples:
    // An implementation of transmit() must be included to simulate
    // an actual transmission of a packet of size "size" over the
    // medium being modelled.
    void transmit(unsigned int size);

    // The function time() must return the duration of the simulated
    // packet transmission. In both TP_general and TP_two_state,
    // this function simply returns the attribute trans_time.
    double time(void);

    // A boolean value is returned by this function to indicate whether
    // packet loss has occurred during transmission. A value of
    // true means that packet loss has occurred. Both TP_general
    // and TP_two_state are lossless transmit processes (i.e. they
    // always return false).

// Private functions
private:
    // Any functions that are internal to this object should be
    // defined as private.

// Protected functions
protected:
    // If there is scope for deriving this class, it would be wise
    // to define some functions as protected, to enable subclasses
    // to have access to them.

};

// Must include this line
#endif

```

## 14.2 Create a .cpp file

The source file must be written to provide implementations for the functions listed in the class definition in `New_Transmit_Process.h` (that haven't already been defined there).

Note: the header file `New_Transmit_Process.h` must be `#included` at the top of the `.cpp` file.

The following is a template of what `New_Transmit_Process.cpp` should look like:

```
// New_Transmit_Process.cpp

#include:  - standard library files.
          - header files from the simulator.
          - any other headers.
          New_Transmit_Process.h

// New_Transmit_Process may need none, some, or all of the following
// global variables which are used in the simulations.

extern Event_list event_list;
extern Node_map node_map;
extern Con_table con_table;

// Definition of the constructor

// Specific example:
// Definition of the constructor that will initialise the Transmit
// Process object and its various attributes.
New_Transmit_Process::New_Transmit_Process(all user-defined parameters)
{
    // Initialise internal variables and attributes.
}

void New_Transmit_Process::transmit(unsigned int size)
{
    // This function will simulate the transmission of a packet over
    // the medium being modelled. The algorithm that will be implemented
    // depends entirely on the characteristics of the Transmit Process
    // with the final goal being the calculation of transmission time.
```

```
}
```

Any other functions which are defined will be used to assist in the implementation of the functions defined in the template.

### 14.3 Modify main.h

The number of input variables used by the New Transmit Process must be specified by adding the following line of code to main.h (anywhere between the lines `#define MAIN_H` and `#endif`):

```
#define NEW_TRANSMIT_PROCESS_ARGS <number of arguments to the constructor>
```

If the number of arguments exceeds the absolute maximum number of arguments (ten), the line:

```
#define MAX_ARGS 10
```

can be altered accordingly.

### 14.4 Modify main.cpp

This file must be modified to allow the New Transmit Process to be used in simulations.

The first addition to the file is the inclusion of the header file `New_Transmit_Process.h` with the following line:

```
#include "New_Transmit_Process.h"
```

This line should be placed anywhere among the lines where other `.h` files are `#included`.

The second addition to the file is the inclusion of source code in the function `add_process()`. The following code must be added before the last `else`-statement:

```
else if(word == "new_transmit_process")
// The string "new_transmit_process" is found in world files that use
// the New Transmit Process.
{
// This obtains all the input arguments for the module from
// the world file.
for(i=0; i<NEW_TRANSMIT_PROCESS_ARGS;i++)
    args[i] = get_next_word(line, &pos);
```

```

// Create a New_Transmit_Process object.
// If any of the arguments to the constructor is a string (filename),
// the value itself in the args array is used without converting
// it, e.g. args[N] is used instead of atoi(args[N].c_str()).
// Note: atof should be used if argument is a floating point value.
New_Transmit_Process *tp = new New_Transmit_Process(atoi(args[0].c_str()),
atoi(args[1].c_str()), ..., atoi(args[N].c_str()));

// Insert a pointer to the object into the trans_procs structure.
trans_procs[atoi(args[0].c_str())] = (Transmit_process*)tp;

// Confirmation that the New Transmit Process was created.
cerr << "Added New Transmit Process: " << atoi(args[0].c_str())
<< endl;
}

```

## 14.5 Modify the Makefile

One minor alteration must be made to the Makefile to ensure that the source code of the New Transmit Process compiles correctly.

On the line beginning with the words "CORE\_OBJ", there is a list of filenames with a .o suffix. The filename of the New Traffic Source with a .o extension added to it must be inserted at the end of the list as shown below:

```
CORE_OBJ = main.o routes.o ... tp_general.o New_Transmit_Process.o
```

## 14.6 Note the changes to the World File syntax

The final step of the process involves adding a new line to the syntax of the world file. This does not need to be done explicitly, but it would demonstrate to any subsequent users how to define the New Transmit Process in a world file. All Transmit Processes must be defined in world files before they are assigned to SLinks.

A line similar to the following one should be included in the section where all other Transmit Processes are defined:

```
process new_transmit_process <id> <arg1> <arg2> ... <argN>
```

where:

**process:** specifies that the New Transmit Process is a Transmit\_process object.

**new\_transmit\_process:** is the string used to identify the New Transmit Process in the world file.

**id:** is the unique id number of the New Transmit Process.

*<arg1> <arg2> ... <argN>*: are user defined arguments required for the implementation of the New Transmit Process. These are the inputs to the New Transmit Process's constructor.