

QNS Reference Manual

Created by: Socrates Toussas, Ala Khanbashi & Dincer Terzioglu

3rd November 2003

Abstract

The Reference Manual contains a detailed explanation of the modules and classes that comprise the QNS. Each section describes a class's #define values, structures, data members and functions. The table of contents below can be used for easier navigation of this part of the documentation.

Contents

1	Class Inheritance Diagram	13
2	Node	14
2.1	Data Members	14
2.1.1	The_id	14
2.1.2	Stopped	14
2.2	Functions	14
2.2.1	Node()	15
2.2.2	In()	15
2.2.3	Out_event()	15
2.2.4	Out()	15
2.2.5	Busy()	15
2.2.6	Start()	15
2.2.7	Stop()	16
2.2.8	Log()	16
2.2.9	Service()	16
2.2.10	Id()	16
2.2.11	Running()	16
2.2.12	Next_dest()	16
2.2.13	Initialise()	16
3	Link	17
3.1	Data Members	17
3.1.1	Buff	17
3.1.2	Time_to_free	17
3.1.3	Rate	17
3.1.4	Prop_delay	17
3.2	Functions	17
3.2.1	Link()	17
3.2.2	Busy()	18
3.2.3	Bandwidth()	18
3.2.4	When_can_send()	18
3.2.5	In()	18
3.2.6	Out()	18
3.2.7	Start()	18
3.2.8	Stop()	18
3.2.9	Log()	19
3.2.10	Service()	19
3.2.1	Delay()	19
3.2.2	Get_next_packet()	19
3.2.3	Queue_packet()	19
3.2.4	In_flight()	19
3.2.5	Set_time_to_free()	19
3.2.1	Serve()	19

4	SLink	20
4.1	Data Members	20
4.1.1	Trans_func	20
4.1.2	Cur_xmit_time	20
4.2	Functions	20
4.2.1	SLink()	20
4.2.2	In()	20
4.2.3	Log()	21
4.2.4	Service()	21
5	Traffic_source	21
5.1	Data Members	21
5.1.1	Buff	21
5.2	Functions	21
5.2.1	Traffic_source()	21
5.2.2	Get()	22
5.2.3	Out()	22
5.2.4	In()	22
5.2.5	Service()	22
5.2.1	Queue_packet()	22
6	TS_Exp	22
6.1	Data Members	23
6.1.1	P_size	23
6.1.2	Sep_mean	23
6.1.3	Output_node	23
6.2	Functions	23
6.2.1	TS_Exp()	23
6.2.2	Get()	23
6.2.3	Next_dest()	24
6.2.4	Log()	24
7	TS_Greedy	24
7.1	Data Members	24
7.1.1	P_size	24
7.1.2	Output_node	24
7.2	Functions	24
7.2.1	TS_Greedy()	24
7.2.2	Get()	25
7.2.3	Next_dest()	25
7.2.4	Log()	25

8	Transmit_process	25
8.1	Functions	25
8.1.1	Transmit()	25
8.1.2	Time()	25
8.1.3	Lost()	25
9	TP_two_state	26
9.1	Structures	26
9.1.1	CHAN_STATE	26
9.2	Data Members	26
9.2.1	Rate_good	26
9.2.2	Rate_bad	26
9.2.3	P_param	26
9.2.4	Q_param	26
9.2.5	Cur_state	26
9.2.6	State_time	26
9.2.7	Trans_time	27
9.3	Functions	27
9.3.1	TP_two_state()	27
9.3.2	Transmit()	27
9.3.3	Time()	27
9.3.4	Lost()	27
10	TP_general	27
10.1	Data Members	28
10.1.1	Cur_state	28
10.1.2	State_time	28
10.1.3	States	28
10.1.4	Trans_time	29
10.2	Functions	29
10.2.1	TP_general()	29
10.2.2	Transmit()	29
10.2.3	Time()	29
10.2.4	Lost()	29
11	Gen_state	29
11.1	Data Members	29
11.1.1	Sojourn_time	29
11.1.2	Cum_prob & Trans_time	30
11.2	Functions	30
11.2.1	Gen_state()	30
11.2.2	Set_sojourn_time()	30
11.2.3	Get_sojourn_time()	30
11.2.4	Last_prob()	30
11.2.5	Add()	30
11.2.6	Get_prob()	30

11.2.7	Get_time()	31
12	Generator	31
12.1	Structures	31
12.1.1	Gen_payload	31
12.2	Data Members	31
12.2.1	Buff	31
12.2.2	Ts	31
12.2.3	W_size	31
12.2.4	In_flight	32
12.2.5	Last_received	32
12.2.6	Last_sent	32
12.2.7	Dest	32
12.3	Functions	32
12.3.1	Generator()	32
12.3.2	In()	32
12.3.3	Out()	33
12.3.4	Log()	33
12.3.5	Service()	33
12.3.6	Start()	33
12.3.1	Dest_node()	33
12.3.2	Get_next_packet()	34
12.3.3	Queue_packet()	34
12.3.4	In_flight()	34
12.3.5	Set_in_flight()	34
12.3.6	Ack_received()	34
12.3.7	Set_w_size()	34
12.3.8	Get_w_size()	34
12.3.9	Get_new_packet()	34
13	TCP	35
13.1	#Defines	35
13.1.1	DEFAULT_AWND_SIZE	35
13.1.2	TCP_INIT_SS_THRESH	35
13.1.3	TCP_RTO_MIN	35
13.1.4	TCP_RTO_MAX	35
13.1.5	TCP_INITIAL_RTO	35
13.1.6	TCP_MAX_DUP_ACKS	35
13.1.7	TCP_SERVICE_INTERVAL	35
13.2	Structures	35
13.2.1	TCP_Payload	35
13.2.2	Tcp_opt	36
13.3	Data Members	37
13.3.1	Tp	37
13.3.2	Pac_snt	37
13.3.3	Pac_rcv	38

13.3.4	Print_trace	38
13.4	Functions	38
13.4.1	TCP()	38
13.4.2	In()	38
13.4.3	Out()	39
13.4.4	Log()	39
13.4.5	Service()	39
13.4.6	Start()	39
13.4.1	Update_w_size()	39
13.4.2	Ack_received()	39
13.4.3	Packet_received()	40
13.4.4	Get_new_packet()	40
13.4.5	Resend_packet()	40
13.4.6	Tcp_rtt_estimator()	40
13.4.7	Tcp_set_rto()	40
13.4.8	Tcp_bound_rto()	41
13.4.9	Print_log()	41
14	Router	41
14.1	#Defines	41
14.1.1	ROUTER_SERVICE_INTERVAL	41
14.1.2	ROUTER_ESTIMATOR_SMOOTH	41
14.2	Data Members	41
14.2.1	Buffer_limit	41
14.2.2	Buff	41
14.2.3	Routing_table	42
14.2.4	Usage_table	42
14.2.5	Drop_table	42
14.3	Functions	42
14.3.1	Router()	42
14.3.2	In()	43
14.3.3	Out_event()	43
14.3.4	Out()	43
14.3.5	Log()	43
14.3.6	Service()	43
14.3.7	Next_dest()	43
14.3.8	Initialise()	43
14.3.1	Next_hop()	44
14.3.2	Update_routing_table()	44
14.3.3	Get_next_packet()	44
14.3.4	Queue_packet()	44
14.3.5	Queue_size()	44
14.3.6	Mem_used()	44
14.3.7	Packets_dropped()	44

15	Vertex	45
15.1	Data Members	45
15.1.1	The_id	45
15.1.2	The_cost	45
15.1.3	The_parent_branch	45
15.2	Functions	45
15.2.1	Vertex()	45
15.2.2	Id()	45
15.2.3	Cost()	45
15.2.4	Parent_branch()	46
15.2.5	Operator	46
16	SFC_Router	46
16.1	#Defines	46
16.1.1	SFC_ROUTER_SERVICE_INTERVAL	46
16.1.2	SFC_ROUTER_QUEUE_AVERAGE_SAMPLES	46
16.1.3	SFC_ROUTER_ESTIMATOR_SMOOTH	46
16.1.4	SFC_INIT_RATE_EST	46
16.2	Structures	46
16.2.1	SFC_Payload	46
16.3	Data Members	47
16.3.1	Max_line_rate	47
16.3.2	SFC_q_min	47
16.3.3	SFC_b_param	47
16.3.4	SFC_chan_rate	47
16.3.5	Pac_deps	47
16.3.6	Queue_samples	47
16.3.7	Queue_acc	47
16.3.8	Mean_queue	48
16.3.9	Data_out	48
16.4	Functions	48
16.4.1	SFC_Router()	48
16.4.2	Out_event()	48
16.4.3	Log()	49
16.4.4	Service()	49
16.4.1	Update_estimators()	49
16.4.2	Calc_pq()	49
17	SFC_Rec	49
17.1	#Defines	49
17.1.1	SFC_REC_SMOOTH_FACTOR	49
17.1.2	SFC_REC_INIT_W_SIZE	49
17.1.3	SFC_REC_BUFFER	50
17.1.4	SFC_REC_RATE_TIME_EST	50
17.2	Data Members	50
17.2.1	Awnd_size	50

17.2.2	Last_tcp_awnd	50
17.2.3	Pac_recs	50
17.2.4	Cur_rate	50
17.2.5	Data_in	50
17.2.6	Sfc_max_delta	50
17.2.7	Sfc_tau	51
17.2.8	Last_pq	51
17.2.9	Rtt	51
17.2.10	Last_ack_num	51
17.2.11	Dup_acks	51
17.2.12	Prime_start	51
17.3	Functions	51
17.3.1	SFC_Rec()	51
17.3.2	In()	52
17.3.3	Log()	52
17.3.4	Service()	52
17.3.1	Forward_packet()	52
17.3.2	Update_estimators()	52
17.3.3	Get_next_packet()	53
17.3.4	Min()	53
17.3.5	Max()	53
18	Event	53
18.1	Structures	53
18.1.1	EVENT_TYPE	53
18.2	Data Members	54
18.2.1	Time	54
18.2.2	Repeat_time	54
18.2.3	Node	54
18.2.4	Type	54
18.2.5	The_data	54
18.3	Functions	55
18.3.1	Event()	55
18.3.2	Execute()	55
18.3.3	Get_time()	55
18.3.4	Get_node()	55
18.3.5	Get_data()	56
18.3.6	Operator	56
19	E_comp	56
19.1	Functions	56
19.1.1	Operator()	56

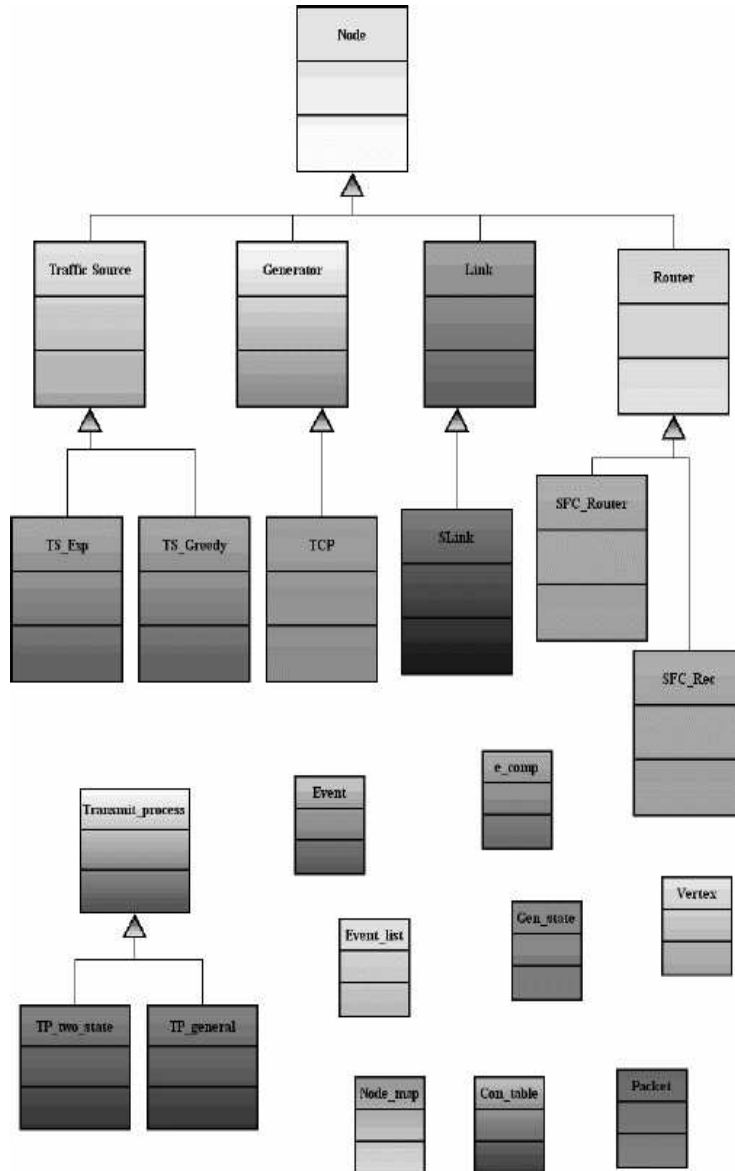
20	Event_list	56
20.1	Data Members	56
20.1.1	Event_q	56
20.1.2	Cur_time;	57
20.2	Functions	57
20.2.1	Event_list()	57
20.2.2	Insert()	57
20.2.3	Get_next()	57
20.2.4	Get_cur_time()	57
20.2.5	Size()	57
21	Main	57
21.1	#Defines	58
21.1.1	MAX_ARGS	58
21.1.2	LINK_ARGS	58
21.1.3	SLINK_ARGS	58
21.1.4	GEN_ARGS	58
21.1.5	TCP_ARGS	58
21.1.6	ROUTER_ARGS	58
21.1.7	SFC_ROUTER_ARGS	58
21.1.8	SFC_FILTER_ARGS	58
21.1.9	TS_GREEDY	58
21.1.10	TS_EXP	58
21.1.11	TP_TWO_STATE	58
21.1.12	DELIM_TOKENS	59
21.1.13	MAX_LINE	59
21.2	Global Variables	59
21.2.1	Event_list	59
21.2.2	Node_map	59
21.2.3	Con_table	59
21.3	Functions	59
21.3.1	Parse()	59
21.3.2	Add_source()	60
21.3.3	Add_process()	60
21.3.4	Build_node()	60
21.3.5	Add_connection()	60
21.3.6	Add_event()	61
21.3.7	Get_next_word()	61
21.3.8	Out_of_store()	61
22	Packet	61
22.1	Structures	61
22.1.1	IPHeader	61
22.1.2	PROTOCOL	62
22.2	Data Members	62
22.2.1	The_header	62

22.2.2	Pac_size	62
22.2.3	Payload	62
22.3	Functions	63
22.3.1	Packet()	63
22.3.2	Size()	63
22.3.3	Header()	63
22.3.4	Set_payload	63
22.3.5	Get_payload()	63
22.3.6	Destination()	63
22.3.7	Source()	63
22.3.8	Protocol()	63
23	Routes	64
24	Node_map	64
24.1	Data Members	64
24.1.1	Nodes	64
24.2	Functions	64
24.2.1	Node_map()	64
24.2.2	Add_node()	64
24.2.3	Id2node()	64
24.2.4	Exists()	64
24.2.5	Num_nodes()	65
24.2.6	Get_all_nodes()	65
25	Con_table	65
25.1	Data Members	65
25.1.1	Connection	65
25.2	Functions	65
25.2.1	Con_table()	65
25.2.2	Add_con()	65
25.2.3	Is_connected()	66
25.2.4	Num_outputs()	66
25.2.5	Get_output()	66
25.2.6	Get_outputs()	66
26	Constants	66
26.1	#Defines	66
26.1.1	BITS_PER_BYTE	66
26.1.2	DEFAULT_TTL	66
26.1.3	ACK_SIZE	66
26.1.4	MAX_DELAY	66

27 ICMP	67
27.1 Structures	67
27.1.1 ICMP_type	67
27.2 Functions	67
27.2.1 Send_ICMP_message()	67

1 Class Inheritance Diagram

The following diagram was created to illustrate the inheritance of the various classes that make up the QNS and to assist in our understanding of how all the modules relate to one another. A triangle indicates inheritance of the class above.



2 Node

The simplest form of network entity, a Node is the basis for all Routers, Generators, Links and Traffic Sources. Each of these objects inherit the basic functionality of the Node and extend it by providing their own implementations for various functions. Node is considered a purely virtual class due to the presence of “= 0” at the end of some of the function prototypes. This means that objects of type Node can only be referenced through a pointer. Each node provides the following basic operations, with the name of the functions that achieve them:

- Process an incoming packet - `in()`
- Output a packet - `out()` and `out_event()`
- Initialisation of node - `initialise()`
- Start or stop receiving packets - `start()` and `stop()`
- Provide status information(e.g. id number, if node is operating and/or available) - `busy()`, `id()` and `running()`
- Output internal state variables - `log()`
- Indication of next destination of output packet - `next_dest()`
- Process periodic service function for each object - `service()`

2.1 Data Members

2.1.1 The `_id`

int the_id;

The unique id number of the Node.

2.1.2 Stopped

bool stopped;

Has the value False if the Node is currently operating.

2.2 Functions

Many functions have been declared virtual (and purely virtual) to allow subclasses to provide their own implementations.

Public Functions

2.2.1 Node()

Node(int i)

The constructor of Node has the following inputs:

i: Unique id number of the Node.

The data member `the_id` is given the value “i” and the stopped data member is set to FALSE to signify that a Node is started upon creation.

2.2.2 In()

virtual void in(Packet) = 0;*

This function is called to process an incoming packet. Each subclass of Node must provide an implementation for this function. Events of type PASS (see Section 18.1.1) must be scheduled to execute this function.

2.2.3 Out_event()

virtual Packet out_event(Event* the_event)*

Only subclasses that require extra information from an event (such as Router (see Section 14.3.2)) should implement this function. The default case simply calls `out()`. Events of type PASS must be scheduled to execute this function.

2.2.4 Out()

virtual Packet out(void) = 0;*

Called to output a packet from a Node. Each subclass of Node must provide an implementation for this function. Events of type PASS (see Section 18.1.1) must be scheduled to execute this function.

2.2.5 Busy()

virtual double busy(void);

Indicates the earliest time that a Node can receive a packet. The default case defined in Node either returns the current time if the Node is operating or an infinite delay (`MAX_DELAY` - see Section 26.1.4) if the Node is not operating.

2.2.6 Start()

virtual void start(void)

A Node starts operating when this function is called. It sets `stopped` to equal FALSE.

2.2.7 Stop()

virtual void stop(void)

A Node stops operating when this function is called. It sets stopped to equal TRUE.

2.2.8 Log()

virtual void log() = 0;

Function that provides logging capabilities to standard output. Most subclasses print statistics during simulations for diagnostic and analytical purposes. Each subclass of Node must provide an implementation for this function. Events of type LOG (see Section 18.1.1) must be scheduled to execute this function.

2.2.9 Service()

virtual void service(void) = 0;

All subclasses that require periodic service functions to be performed on them must provide an implementation of this function. All other subclasses should provide an empty function. Events of type SERVICE (see Section 18.1.1) must be scheduled to execute this function.

2.2.10 Id()

unsigned int id(void)

Returns the id number (the_id) of a Node.

2.2.11 Running()

bool running(void)

Returns TRUE if the Node is running, FALSE if it's not.

2.2.12 Next_dest()

virtual Node next_dest(Event* the_event);*

Returns a pointer to the Node a packet will go to when it is output.

2.2.13 Initialise()

virtual void initialise(void)

The initialisation function for the Node is called before the first event is executed. Primarily used by Routers (see Section 14.3.8) to construct routing tables. Not implemented for the basic Node.

3 Link

The object that connects either routers or generators together in a network is the Link. A Link cannot be connected to another Link (or any other type of link), nor can it be connected to a Traffic Source. A Link is basically modelled as a queue of finite length, and the time each packet spends in the queue is equivalent to the time it would take the same packet to travel along the Link.

3.1 Data Members

3.1.1 Buff

queue<Packet> buff;*

The queue that represents a link. The time spent by packets in this container depends on the size of the packet (in bits), the transmission rate and the propagation delay of the link.

3.1.2 Time_to_free

double time_to_free;

Variable that specifies the earliest time at which the link is available.

3.1.3 Rate

double rate;

The transmission rate of the link in bits/second.

3.1.4 Prop_delay

double prop_delay;

The total propagation delay of the link in seconds.

3.2 Functions

Many functions have been declared virtual to allow subclasses of Link to provide their own implementations.

Public Functions

3.2.1 Link()

Link(const int i, const double ra, const double del)

The constructor of Link has the following inputs:

- i:** Unique id number of the Link node.
- ra:** The transmission rate (bandwidth) of the link.
- del:** The total propagation delay of the link.

The initialisation list in the constructor gives the Node object within the Link object the same id number. The rate and prop_delay data members are

initialised here and the function `set_time_to_free()` (see Section 3.2.5) is also called to initialise the `time_to_free` data member.

3.2.2 Busy()

virtual double busy(void)

Returns the time at which the link will be free next. This value could be either `time_to_free` or the current time (if it's not busy). If the link is not operating, an infinite time (`MAX_DELAY`) is returned to denote this.

3.2.3 Bandwidth()

double bandwidth(void)

Returns the transmission rate, or bandwidth, of the link.

3.2.4 When_can_send()

double when_can_send(int dst)

The definition of this function is not specified.

3.2.5 In()

*virtual void in(Packet *p)*

A packet has arrived at the beginning of the link and must be stored in the buffer. The various parameters that determine the time the packet will take to reach the other side are calculated and an event (see Section 18.3.1) is scheduled to output the packet from the buffer after this time has elapsed.

3.2.6 Out()

virtual Packet out(void)*

The `out()` function is called by the event handler(see Section 18.3.2) to remove a packet from the buffer after the required amount of time has elapsed. A call to `get_next_packet()` (see Section 3.2.2) is made to actually remove the packet.

3.2.7 Start()

virtual void start(void)

This function enables the link to begin receiving packets at its input. The default `start()` function from `Node` is called here.

3.2.8 Stop()

virtual void stop(void)

Prevents the link from receiving packets at its input. The default `stop()` function from `Node` is executed here.

3.2.9 Log()

virtual void log(void)

Various state variables are appended to an output string stream object (ostringstream) where they are then printed to standard output.

3.2.10 Service()

virtual void service(void)

This function is not defined as a Link does not require periodic servicing.

Protected Functions

3.2.1 Delay()

*double delay(Packet *p)*

Returns the value of the prop_delay data member.

3.2.2 Get_next_packet()

Packet get_next_packet(void)*

Called from within out() to physically remove the next packet from the buffer for transmission.

3.2.3 Queue_packet()

*void queue_packet(Packet *p)*

Called from within in(), this function places the new packet into the buffer for later transmission.

3.2.4 In_flight()

int in_flight(void)

Returns the number of packets that are currently in the buffer (i.e. that are currently travelling down the link).

3.2.5 Set_time_to_free()

void set_time_to_free(double time)

Sets the time_to_free data member to equal "time".

Private Functions

3.2.1 Serve()

*double serve(Packet *p)*

The service time of a packet (time taken for packet to travel from one end to the other, not including delays) is calculated by this function. The algorithm

that does the calculation is as follows:

$$(size_of_packet \text{ (in bytes)} * \# \text{ bits per byte}) / bandwidth \text{ (in bits/sec)}$$

4 SLink

The Stochastic Link extends the definition of a Link (see Section 3) by modelling the delay and loss probability as a function of a packet's size. A Transmit Process (see Section 8) object associated with each SLink generates a random delay and loss rate according to the size of each individual packet. The bandwidth of the SLink is also determined by the Transmit Process.

4.1 Data Members

4.1.1 Trans_func

*Transmit_process *trans_func;*

A pointer to the single Transmit Process object associated with the SLink. A Transmit Process simulates the actual transmission of each packet down the SLink.

4.1.2 Cur_xmit_time

double cur_xmit_time;

The time taken to transmit an entire packet over an SLink. The value is obtained by calling the Transmit Process's time() function.

4.2 Functions

Public Functions

4.2.1 SLink()

*SLink(const int i, const double del, const Transmit_process *pt)*

The constructor of SLink has the following input parameters:

- i:** Unique id number of the SLink node.
- del:** Total propagation delay of the SLink.
- pt:** Transmit Process associated with this SLink.

The id and prop_delay of the Link object within the SLink are set to be identical to the SLink's via an initialisation list, but the bandwidth of the Link is set to 0 (Transmit Process determines this). The data member trans_func is set to point to the same Transmit Process that "pt" points to.

4.2.2 In()

*void in(Packet *p)*

A packet's transmission down the SLink is simulated by the `trans_func` object (by executing its `transmit()` function - see Section 8.1.1)). The object also calculates the transmission time which determines the amount of time the packet spends in the SLink's buffer (assuming packet loss did not occur).

4.2.3 `Log()`

void log(void)

An output string stream (`ostringstream`) object is created to hold certain state variables that are printed to standard output.

4.2.4 `Service()`

void service(void)

SLinks, like Links, do not require any periodic servicing, therefore this function is not defined.

5 `Traffic_source`

This abstract class is the basis for all Traffic Sources (that create traffic in the form of packets) in the QNS. A Traffic Source is a type of Node (see Section 2) so this class provides definitions for the functions `in()`, `out()` and `service()`. The functions `in()` and `service()` are not implemented by Traffic Sources and are therefore left empty. Due to the declaration of `get()` (see Section 5.2.2) as a purely virtual function and the absence of the `log()` function, these are the only two functions that all Traffic Sources must have definitions for.

A packet is generated in response to a call to `get()` (by a generator) and depending on the type of Traffic Source, `get()` will determine when the packet will be ready. The derived classes `TS_Greedy` (see Section 7) and `TS_Exp` (see Section 6) differ slightly in this regard.

5.1 Data Members

5.1.1 `Buff`

queue<Packet> buff;*

This is the queue that holds all the packets to be transmitted by the Traffic Source.

5.2 Functions

Public Functions

5.2.1 `Traffic_source()`

Traffic_source(const int i)

The constructor of `Traffic_source` has the following inputs:

i: Unique id number of the Traffic_source node.

An initialisation list simultaneously initialises a Node (see Section 2) object with the same id value. No other initialisations are performed within the constructor.

5.2.2 Get()

virtual void get(int dest) = 0

A purely virtual function that has to be implemented in derived classes. It will create a packet and schedule the time at which the packet will be ready to output. This function is implemented by the two subclasses TS_Greedy (see Section 7.2.2) and TS_Exp (see Section 6.2.2).

5.2.3 Out()

Packet out(void)*

This function returns the packet at the head of the Traffic Source's output buffer.

5.2.4 In()

void in(Packet)*

Packets do not arrive as input to a Traffic Source, therefore this function has not been implemented and has been left empty.

5.2.5 Service()

void service(void)

Traffic Sources do not require any servicing functions to be applied, therefore this function has not been implemented and has been left empty.

Protected Functions

5.2.1 Queue_packet()

*void queue_packet(Packet *p)*

Puts newly created packets onto the output buffer. It is defined as a protected function to restrict access to friend and derived classes only.

6 TS_Exp

A type of Traffic Source (see Section 5) (i.e. a subclass of Traffic_source), TS_Exp simulates packet creation at an exponentially distributed rate. It schedules packets to be output with a time offset which is calculated from an exponentially distributed value.

6.1 Data Members

6.1.1 P_size

int p_size;

This is the packet size, which is specified as one of the inputs of the constructor.

6.1.2 Sep_mean

double sep_mean;

This is the mean packet separation time, which is used to calculate the exponentially derived offset.

6.1.3 Output_node

unsigned int output_node;

Variable that contains the id of the generator that receives packets from the Traffic Source.

6.2 Functions

Public Functions

6.2.1 TS_Exp()

TS_Exp(const int i, int s, double is)

The constructor of TS_Exp has the following input parameters:

- i:** Unique id number of the TS_Exp node.
- s:** Packet size.
- is:** Mean packet separation time.

An initialisation list in the constructor simultaneously initialises a Traffic Source (see Section 5) object with the same id value of “i”. The constructor also initialises the value of p_size and sep_mean to the given values of “s” and “is” respectively.

6.2.2 Get()

void get(const int dest)

This function determines the rate at which the exponential Traffic Source outputs packets. After the creation of a packet, an event of type PASS (see Section 18.1.1) is scheduled to indicate that the packet will be output at a later time (not immediately). The time is an exponentially calculated offset to the current time.

6.2.3 Next_dest()

Node next_dest(Event* the_event)*

This function will return a pointer to the generator that the exponential Traffic Source is attached to.

6.2.4 Log()

void log(void)

This function has not been implemented. It can provide a printout of statistical information.

7 TS_Greedy

This is a subclass of Traffic Source (see Section 5). It simulates a simple greedy Traffic Source where each packet is output immediately after creation.

7.1 Data Members

7.1.1 P_size

int p_size;

This is the packet size in bytes. It is initialised by the constructor.

7.1.2 Output_node

unsigned int output_node;

This is the id of the generator that receives packets from the greedy Traffic Source.

7.2 Functions

Public Functions

7.2.1 TS_Greedy()

TS_Greedy(const int i, int s)

The constructor of TS_Greedy has the following inputs:

- i:** Unique id number of the TS_Greedy node.
- s:** Packet size.

Whenever a TS_Greedy object is created, an initialisation list in the constructor simultaneously initialises a Traffic_source object with the same id number “i”. The constructor also initialises the value of p_size to the given value “s”.

7.2.2 Get()

void get(const int dest)

This function determines the rate at which the greedy Traffic Source outputs packets. After the creation of a packet, an event of type PASS (see Section 18.1.1) is scheduled to indicate that the packet will be output immediately.

7.2.3 Next_dest()

Node next_dest(Event* the_event)*

This function returns a pointer to the generator that the greedy Traffic Source is attached to.

7.2.4 Log()

void log(void)

The function that prints out statistical information is not implemented.

8 Transmit_process

This is an abstract base class which is used to model the transmission of packets over a medium (an SLink - see Section 4). A Transmit Process must implement the three functions below.

8.1 Functions

Public Functions

8.1.1 Transmit()

virtual void transmit(unsigned int size) = 0

This function simulates the actual transmission of a packet of size “size”.

8.1.2 Time()

virtual double time(void) = 0

This returns the time it took to transmit the packet.

8.1.3 Lost()

virtual bool lost(void) = 0

Returns true if the packet was lost, and false otherwise.

9 TP_two_state

This class simulates a two state channel Transmit Process that is lossless. Since a packet transmission retains state, a single two_state transmission process should be used per link.

9.1 Structures

9.1.1 CHAN_STATE

The following enumerator keeps a record of channel transmission state (either GOOD or BAD).

```
typedef enum CHAN_STATE {  
    GOOD,  
    BAD  
};
```

9.2 Data Members

9.2.1 Rate_good

unsigned int rate_good;

The rate of transmission in the GOOD state.

9.2.2 Rate_bad

unsigned int rate_bad;

The rate of transmission in the BAD state.

9.2.3 P_param

double p_param;

The rate at which the transmission process leaves the GOOD state.

9.2.4 Q_param

double q_param;

The rate at which the transmission process leaves the BAD state.

9.2.5 Cur_state

CHAN_STATE cur_state;

The channel state.

9.2.6 State_time

double state_time;

The time at which the current state was recorded.

9.2.7 Trans_time

double trans_time;

The transmission time of the last packet.

9.3 Functions

Public Functions

9.3.1 TP_two_state()

TP_two_state(unsigned int ra1, unsigned int ra2, double p, double q)

The constructor of TP_two_state has the following input parameters:

- ra1:** The rate of transmission in the GOOD state.
- ra2:** The rate of transmission in the BAD state.
- p:** The rate at which the transmission process leaves the GOOD state.
- q:** The rate at which the transmission process leaves the BAD state.

The constructor basically sets the attributes of the class to the values given and initialises the channel state .

9.3.2 Transmit()

void transmit(unsigned int size)

Transmission of a packet of size “size” is handled by this function. The number of bits that can be transmitted in the current state is calculated. If the whole packet cannot be transmitted in this state, the remaining portion is transmitted in the other state.

E.g. if x number of bits (where $x < \text{total number of bits to send}$) can be transmitted in the GOOD state, the remaining bits ($\text{total number of bits} - x$) will be transmitted in the BAD state.

9.3.3 Time()

double time(void)

Returns the length of time it took to transmit the packet.

9.3.4 Lost()

bool lost(void)

Returns true if the packet was lost and false otherwise.

10 TP_general

Simulates a general packet Transmit Process that is lossless. Packet transmission retains state, so only one general Transmit Process should be used per link. The general transmission method follows the information contained in two files,

they are:

State specification file

This file contains the data which outlines the behaviour of a process and how it moves between states. The syntax of the state specification file is as follows:

```
<time_spent_in_state> <file_containing_cmf>
...
...
EOF
```

The state is characterised by a file containing the cumulative mass function (cmf). The syntax of the cmf file is as follows:

```
<time_to_transmit> <cumulative probability>
...
...
EOF
```

The process will cycle through these states according to the values in the two files. Each state specification file is associated with an instance of the class `Gen_state` (see Section 11) that stores the data from the file containing the cmf.

10.1 Data Members

10.1.1 `Cur_state`

```
unsigned int cur_state;
```

The current state of the Transmit Process.

10.1.2 `State_time`

```
double state_time;
```

The last time the state was recorded.

10.1.3 `States`

```
vector<Gen_state*> states;
```

This is effectively a vector of state descriptions. Each state consists of a cumulative density function (of transmission times) which is then used to generate packet transmission times.

e.g. `states[1]` = pairs of values describing a state where:
value.first = cumulative probability
value.second = transmission time

10.1.4 Trans_time

double trans_time;

Transmission time of last packet.

10.2 Functions

Public Functions

10.2.1 TP_general()

TP_general(string sf)

The constructor of TP_general has the following inputs:

sf: Name of state specification file.

The constructor opens the state specification file and constructs the necessary Gen_state objects.

10.2.2 Transmit()

void transmit(unsigned int size)

Simulation of the transmission of a packet of size “size” occurs within this function. The calculation of transmission time is based on the specifications in the state description file and the file containing the cmf.

10.2.3 Time()

double time(void)

Returns the transmission time of the packet of size “size”.

10.2.4 Lost()

bool lost(void)

Returns true if the packet was lost, and false otherwise.

11 Gen_state

This class is used by TP_general to store information regarding the state and cumulative mass functions and their corresponding transmission times. It is defined in the same source file as TP_general.

11.1 Data Members

11.1.1 Sojourn_time

double sojourn_time;

Indicates the length of time to stay in the particular state.

11.1.2 Cum_prob & Trans_time

```
vector<double> cum_prob;  
vector<double> trans_time;
```

These two vectors hold the cumulative mass functions and the corresponding transmission times.

11.2 Functions

Public Functions

11.2.1 Gen_state()

```
Gen_state(void)
```

The constructor doesn't perform any operations or initialisations. The class's main purpose is for storage.

11.2.2 Set_sojourn_time()

```
void set_sojourn_time(double t)
```

Sets the `sojourn_time` data member.

11.2.3 Get_sojourn_time()

```
double get_sojourn_time(void)
```

Returns the value of `sojourn_time`.

11.2.4 Last_prob()

```
double last_prob(void)
```

Returns the last cumulative probability.

11.2.5 Add()

```
void add(double time, double prob)
```

This function inserts the variable "time" at the end of the vector `trans_time` and inserts "prob" at the end of the vector `cum_prob`. It basically constructs the cumulative mass function and corresponding transmission times.

11.2.6 Get_prob()

```
double get_prob(unsigned int i)
```

Returns the probability value at a particular position "i" in the "cum_prob" vector.

11.2.7 Get_time()

double get_time(unsigned int i)

Returns the transmission time value from the trans_time vector at the position “i”.

12 Generator

A Generator is a node that generates and/or absorbs packets in a network using window flow control. A Generator uses the simplest form of flow control, where the window size remains fixed throughout the lifetime of the simulation and is not altered according to any algorithm. A TCP node (see Section 13) is derived from a Generator, and it is possible to derive other, more complex nodes that inherit and extend the basic functionality of a Generator.

12.1 Structures

12.1.1 Gen_payload

The header for packets generated and absorbed by a Generator contains important information about the packet itself and has the following specification:

```
typedef struct {
    unsigned int seq_no;           // sequence number of packet
    bool         ack;             // is it an ack or payload?
    void         *the_payload;    // data attached to it
} Gen_payload
```

12.2 Data Members

12.2.1 Buff

queue<Packet> buff;*

The output buffer of a Generator holds packets that are waiting to be transmitted.

12.2.2 Ts

*Traffic_source *ts;*

A pointer to a Traffic Source (see Section 5) which supplies new packets to the Generator. A Traffic Source must be referenced through a pointer because it is specified as a purely abstract class (due to the presence of “= 0” at the end of the declaration of a virtual function).

12.2.3 W_size

double w_size;

The current size of the transmission window.

12.2.4 In_flight

unsigned int in_flight;

The current number of packets in the network that originated from this Generator.

12.2.5 Last_received

unsigned int last_received;

The sequence number of the last ack that was received (or the sequence number of the last packet that was confirmed to be received by another node).

12.2.6 Last_sent

unsigned int last_sent;

The sequence number of the packet that was last sent from this Generator.

12.2.7 Dest

unsigned int dest;

The id of the node that the Generator is connected to.

12.3 Functions

Public Functions

12.3.1 Generator()

*Generator(const int id, const int d, const int w, Traffic_source *s)*

The constructor of Generator has the following input parameters:

- id:** Unique id number of Generator node.
- d:** Id number of node that Generator is connected to.
- w:** Window size of the Generator.
- *s:** Pointer to Traffic Source connected to Generator.

The constructor simultaneously initialises a Node (see Section 2) object with the same id number as the Generator. Within the constructor, the data members `w_size`, `ts`, `inflight`, `last_received` and `dest` are initialised.

12.3.2 In()

*virtual void in(Packet *p)*

This function is called as a result of a PASS event (see Section 18.1.1) being scheduled by a node connected to a Generator. It's purpose is to process an incoming packet, which must only have RAW (see Section 22.1.2) or GENERATOR (see Section 22.1.2) in its protocol field, and whose destination is the Generator itself. If any of these conditions are violated, an error has occurred in the simulation.

A RAW packet is a newly created packet from a Traffic Source. The Generator creates and attaches a header (`Gen_payload`) to it and schedules an event to output the packet as soon as possible.

GENERATOR Packets originate from other Generators and they contain either an acknowledgement (`ack`) or data in their payload section. If the packet contains an `ack`, the function `ack_received()` (see Section 12.3.6) is executed to update state variables and an attempt is also made to generate new packets into the network. If the packet contains data, an `ack` packet is created and sent to the Generator that transmitted the data packet.

12.3.3 `Out()`

virtual Packet out(void)*

An event of type `PASS` (see Section 18.1.1) calls this function to output a packet from the Generator. The function `get_next_packet()` (see Section 12.3.2) is executed to physically remove the packet from the output buffer for sending.

12.3.4 `Log()`

virtual void log(void)

An output string stream (`ostringstream`) object is created and various state variables are appended to it. The `ostringstream`'s contents are then printed to standard output along with the Generator's id and the current time in the simulation.

12.3.5 `Service()`

virtual void service(void)

Since a Generator does not schedule `SERVICE` events (see Section 18.1.1), this function is not defined (it is left empty).

12.3.6 `Start()`

virtual void start(void)

The Node class' `start()` (see Section 2.2.6) function is executed and `get_new_packet()` (see Section 12.3.9) is called repeatedly to generate new packets from the attached Traffic Source.

Protected Functions

12.3.1 `Dest_node()`

unsigned int dest_node(void)

Returns the id of the node that the Generator is connected to (the next hop).

12.3.2 Get_next_packet()

Packet get_next_packet(void)*

This function obtains the next packet for transmission from the output buffer.

12.3.3 Queue_packet()

*void queue_packet(Packet *p)*

A packet is placed on the end of the output buffer.

12.3.4 In_flight()

int in_flight(void)

Returns the value of the inflight variable.

12.3.5 Set_in_flight()

void set_in_flight(unsigned int i)

Sets the inflight variable.

12.3.6 Ack_received()

void ack_received(unsigned int seq_no)

Updates the inflight and last_received variables whenever an ack has been received. For this function to work, it must be assumed that packets can never be reordered if they are of the same flow.

12.3.7 Set_w_size()

void set_w_size(double w)

This function is not defined or implemented.

12.3.8 Get_w_size()

double get_w_size(void)

This function is not defined or implemented.

12.3.9 Get_new_packet()

void get_new_packet(void);

The Generator makes a request for a new packet from the attached Traffic Source by executing the Traffic Source's get() (see Section 5.2.2) function. The Traffic Source must create a new packet and schedule an event to pass it to the Generator.

13 TCP

A generator operating under the Transmission Control Protocol (TCP), a TCP node uses a more complex form of window flow control where the send and receive windows continuously vary in size.

13.1 #Defines

13.1.1 `DEFAULT_AWND_SIZE`

The default advertised window size.

13.1.2 `TCP_INIT_SS_THRESH`

The initial Slow Start threshold value.

13.1.3 `TCP_RTO_MIN`

The minimum Retransmission Time Out (RTO) time.

13.1.4 `TCP_RTO_MAX`

The maximum RTO time.

13.1.5 `TCP_INITIAL_RTO`

The initial RTO time.

13.1.6 `TCP_MAX_DUP_ACKS`

Maximum number of duplicate ACKs to indicate a packet loss.

13.1.7 `TCP_SERVICE_INTERVAL`

Interval where TCP checks for time outs.

13.2 Structures

13.2.1 `TCP_Payload`

The header for packets generated and absorbed by a TCP node contains important packet related information and has the following specification:

```

typedef struct {
    unsigned int      src_port,      dest_port;
    unsigned long     seq_no;
    unsigned long     ack_no;
    // now the flags
    bool              urg, ack, psh,  rst, syn, fin;
    unsigned int      win_size;
    unsigned int      check_sum;
    unsigned int      urgent_ptr;
    // time stamp option
    double            time_stamp;
    void              *_the_payload; // data attached to the packet
} TCP_Payload;

```

13.2.2 Tcp_opt

The structure which contains all the TCP state variables that simulate the main features of the protocol. It is a small subset of the structure defined in sock.h from the ipv4 net implementation in the linux kernel, also called tcp_opt. The following is the specification of the structure:

```

// A small subset of tcp_opt defined in sock.h from the ipv4 net implementation
// in the linux kernel.
struct tcp_opt {

```

```

unsigned long      rcv_next;           // seq_no we want to receive next
unsigned long      snd_next;           // seq_no we want to send next
unsigned long      last_ack;           // seq_no of last ACK
// RTT measurement
unsigned int       backoff;            // backoff
double            srtt;                // smoothed round trip time << 3
double            mdev;                // medium deviation
double            mdev_max;            // maximal mdev for the last rtt period
double            rttvar;              // smoothed mdev_max
unsigned long      rtt_seq;            // sequence number to update rttvar
double            rto;                 // retransmit timeout
double            last_ack_received;   // time last ack was received
double            last_rtt;            // save last rtt for logging purposes ONLY
// not actually used in TCP function

unsigned long      packets_out;        // Bytes which are "in flight"
unsigned long      left_out;           // Packets which leaved network
unsigned long      retrans_out;        // Retransmitted packets out
unsigned int       mss;                // maximum segment size in bytes
unsigned long      snd_ssthresh;       // Slow start size threshold
unsigned long      snd_cwnd;           // Sending congestion window
unsigned long      snd_cwnd_cnt;       // Linear increase counter
unsigned long      snd_cwnd_clamp;     // Do not allow snd_cwnd to grow above this
unsigned long      snd_cwnd_used;
unsigned long      snd_cwnd_stamp;
unsigned long      seq_no_when_loss;   // the sequence number when a loss occurred
unsigned long      rcv_wnd;            // current receiver window (awnd)
unsigned long      window_clamp;       // Maximum window to advertise
unsigned int       keepalive_time;     // time before keep alive takes place
unsigned int       keepalive_intvl;    // time interval between keep alive probes
unsigned int       dup_acks;           // number of duplicate acks received
};

```

13.3 Data Members

13.3.1 Tp

tcp_opt tp;

Structure of type `tcp_opt` that holds all TCP state variables.

13.3.2 Pac_snt

map<unsigned long, unsigned int> pac_snt;

Container that holds the packets that have been sent, but not yet acknowledged.

Key (unsigned long): Sequence number of packet.

Data (unsigned int): Size of packet.

13.3.3 Pac_rcv

set<unsigned long> *pac_rcv*;

Ordered buffer of packets that have been received out of sequence.

Data: Sequence number of packet.

13.3.4 Print_trace

bool print_trace;

Boolean to specify whether to log packet arrivals.

13.4 Functions

Public Functions

13.4.1 TCP()

*TCP(const unsigned int id, const unsigned int d, const unsigned int mss, Traffic_source *s)*

The constructor of TCP has the following inputs:

- id:** Unique id number of TCP node.
- d:** Id number of node that TCP node is connected to.
- mss:** Maximum segment size.
- *s:** Pointer to Traffic Source connected to TCP node.

When a TCP object is created, the constructor simultaneously initialises a Generator (see Section 12) object. The Generator object is given the same values of id, d and s, as well as w (window size) equal to 1. Within the constructor, many of the TCP state variables (defined in *tcp_opt*) are initialised.

13.4.2 In()

*void in(Packet *p)*

Packets arriving at a TCP node must have either RAW (see Section 22.1.2) or TCP (see Section 22.1.2) in the protocol field, and the TCP node as the destination. If either of these is not the case, an error has occurred in the simulation.

A RAW packet has recently been created from a Traffic Source (see Section 5). The TCP node creates and attaches a header (*TCP_payload*) to it and schedules an event to output the packet as soon as possible.

Packets with TCP in the protocol field originate from other TCP nodes and they contain either an acknowledgement (ack) or data in their payload (header) section. If the packet contains an ack, the function *ack_received()* (see Section 13.4.2) is executed to determine whether it was the expected one and to update the TCP state variables (e.g. window size). If the window size permits, the function *get_new_packet()* (see Section 13.4.4) is called to generate a packet into the network. If the packet contains data the function *packet_received()*

(see Section 13.4.3) is called to determine if it was the expected one. An ack packet is created and sent to the TCP node that transmitted the data packet.

13.4.3 Out()

Packet out(void)*

An event of type PASS (see Section 18.1.1) calls this function to output a packet from the TCP node. The function `get_next_packet()` (see Section 12.3.2), defined in the Generator class, is executed to physically remove the packet from the output buffer for sending. Data packets are given time-stamps.

13.4.4 Log()

void log(void)

This function is called to activate logging of packet arrivals for diagnostics purposes. It basically sets the `print_trace` (see Section 13.3.4) boolean to TRUE.

13.4.5 Service()

void service(void)

The main task of this function is to check for time outs of acks. If the ack for the last transmitted packet hasn't arrived, the RTO time is examined. If the RTO time has expired, a time out has occurred and packet loss is assumed. The RTO time is then doubled and the lost packet is retransmitted by executing `resend_packet()` (see Section 13.4.5).

13.4.6 Start()

void start(void)

The Node class' `start()` (see Section 2.2.6) function is executed to begin operation. A SERVICE event (see Section ??) is then scheduled to begin the periodic task defined in `service()` and `get_new_packet()` is called to begin generating packets.

Private Functions

13.4.1 Update_w_size()

void update_w_size(void)

The size of the sending congestion window is increased to avoid packet congestion in the network.

13.4.2 Ack_received()

*void ack_received(TCP_Payload *pl)*

This function determines the following:

- If the ack is a duplicate, it determines whether packet loss has occurred or if the node is still in fast recovery/retransmit mode and then performs the necessary operations.
- If the ack is not a duplicate, a sent packet has been acknowledged and is removed from the `pac_snt` (see Section 13.3.2) buffer. Various TCP state variables and RTO values are also updated.

13.4.3 Packet_received()

unsigned int packet_received(unsigned int seq_no)

This function determines the sequence number of the packet that will be acknowledged. This could be the number of the last packet received (packet was duplicate and packet loss assumed), or the number of the current packet received (packet was received in order). Packets received out of order are also handled by this function.

13.4.4 Get_new_packet()

void get_new_packet(void)

A request is made for a new packet from the Traffic Source by calling the Generator class' `get_new_packet()` (see Section 12.3.9) function which in turn executes the Traffic Source's `get()` (see Section 5.2.2) function. The Traffic Source must create a new packet and schedule an event to pass it onto the TCP node.

13.4.5 Resend_packet()

void resend_packet(unsigned long seq_no)

This function is called when packet loss has occurred due to timed out acks or too many duplicate acks. A new packet is created with a header (`TCP_payload`) attached to it and an event is scheduled to output the packet as soon as possible.

13.4.6 Tcp_rtt_estimator()

void tcp_rtt_estimator(double mrtt)

This function was adapted from the ipv4 inet implementation of the Linux kernel release 2.4.3 to make the simulation more accurate when calculating and updating the various RTT related variables. It is called whenever an ack is received.

13.4.7 Tcp_set_rto()

void tcp_set_rto(void)

This function was adapted from the ipv4 inet implementation of the Linux kernel release 2.4.3 to make the simulation more accurate when calculating and updating the various RTO related variables. It is called whenever an ack is received.

13.4.8 `Tcp_bound_rto()`

void tcp_bound_rto(void)

This function was adapted from the ipv4 inet implementation of the Linux kernel release 2.4.3 to make the simulation more accurate when calculating and updating the various RTO related variables. It is called whenever an ack is received.

13.4.9 `Print_log()`

void print_log(void)

This function provides statistics when packets arrive at a TCP node. An output string stream (ostringstream) object is created and the statistics are appended to it. The ostream's contents are then printed to standard output along with the node's id and the current time in the simulation.

14 Router

A Router in a network ensures that all packets that arrive at its input are directed onto the correct output port (link). All packets that pass through a Router must be buffered at the appropriate queue before being transmitted. A separate queue is maintained for each output port.

14.1 `#Defines`

14.1.1 `ROUTER_SERVICE_INTERVAL`

The time interval that the Router periodically services itself (i.e. how often `service()` is executed).

14.1.2 `ROUTER_ESTIMATOR_SMOOTH`

Exactly one tenth of the Router Service Interval. Not utilised in this class.

14.2 Data Members

14.2.1 `Buffer_limit`

unsigned int buffer_limit;

A user defined parameter that specifies the maximum buffer size for each output port.

14.2.2 `Buff`

map<int, queue<Packet> > buff;*

The container that holds the buffers for each output port.

Key (int): Output port number. Same as id of link connected to it.

Data (queue<Packet*>): Buffer for each output port.

14.2.3 Routing_table

map<int, int> routing_table;

The routing table maintains information about the connection of nodes in a network. These connections describe the cheapest paths that packets can take to get to their destination.

Key (int): A vertex (node) in the network.

Data (int): The parent branch of the vertex (cheapest way to get to vertex).

14.2.4 Usage_table

map<unsigned int, unsigned int> usage_table;

Maintains information about the usage of all queues in the router, i.e. the total size of all packets in every output buffer.

Key (unsigned int): Output queue (port) number.

Data (unsigned int): Size of the queue.

14.2.5 Drop_table

map<unsigned int, unsigned long> drop_table;

The drop table keeps track of the number of packets that each output queue has dropped. A packet is dropped if it arrives at a full queue.

Key (unsigned int): Output queue (port) number.

Data (unsigned int): Number of packets dropped.

14.3 Functions

Public Functions

14.3.1 Router()

Router(const int i, const unsigned int bl)

The constructor of Router has the following input parameters:

i: Unique id number of Router node.

bl: Maximum size of an output queue for this Router.

The constructor sets the buffer limit and then schedules a repeating SERVICE event (see Section 18.1.1) to repeatedly execute the service() function. The Node (see Section 2) object within the Router object is also given the same id number by way of an initialisation list.

14.3.2 In()

*virtual void in(Packet *p)*

This function determines the correct output port that each packet must be routed to and places the packet onto the relevant queue (if the queue is full the packet is dropped). Each PASS event (see Section 18.1.1) scheduled by the Router includes extra information (pointer to int) to indicate which queue the packet is output from. A Router also ensures that packets never endlessly travel across a network by discarding those with Time To Live (TTL) fields equal to zero.

14.3.3 Out_event()

virtual Packet out_event(Event *the_event)*

The input parameter "the_event" contains data that specifies which port a packet exits the Router from. The function get_next_packet() (see Section 14.3.3) physically removes the packet from the relevant queue.

14.3.4 Out()

virtual Packet out(void)*

This function is not implemented by a Router, as out_event() is used instead.

14.3.5 Log()

virtual void log(void)

Various statistics are appended to an output string stream (ostream) object and then printed to standard output for analysis. The results of the function mem_used() (see Section 14.3.6) are used here.

14.3.6 Service()

virtual void service(void)

A Router does not require regular servicing, therefore this function is not defined.

14.3.7 Next_dest()

Node next_dest(Event *the_event)*

Returns a pointer to the node (Link) connected to the output port that a packet leaves from. The extra data in "the_event" specifies the correct output port number.

14.3.8 Initialise()

virtual void initialise(void)

Performs initial tasks for the Router before the first event is executed. These tasks consist of creating a static routing table by calling update_routing_table()

and initialising the `usage_table` (see Section 14.2.4) and `drop_table` (see Section 14.2.5) data members.

Protected Functions

14.3.1 Next_hop()

int next_hop(const unsigned int dst)

Searches the routing table to determine the next hop that a packet must take to reach its destination.

14.3.2 Update_routing_table()

void update_routing_table(void)

This function creates a static routing table using data from the `Con_table` (see Section 25) (connection) class. Dijkstra's Shortest Path Algorithm utilises a set of `Vertex` (see Section 15) objects to find the optimal paths through the network. These optimal paths are then stored in the routing table.

14.3.3 Get_next_packet()

Packet get_next_packet(Event *the_event)*

A packet is removed from the relevant output queue (as determined by "the_event") for sending and the `usage_table` is updated.

14.3.4 Queue_packet()

void queue_packet(Packet p, int the_queue)*

The packet is inserted into the relevant queue and the `usage_table` is updated.

14.3.5 Queue_size()

unsigned int queue_size(unsigned int q)

Returns the usage (or size) of the queue specified by "q".

14.3.6 Mem_used()

unsigned int mem_used(void)

Returns the total usage (or size) of all queues in the Router.

14.3.7 Packets_dropped()

unsigned long packets_dropped(void)

Returns the total number of packets dropped by the Router during the simulation.

15 Vertex

A set of these objects in a Router (see Section 14) represents nodes in the network, that have had their shortest path determined, for the purpose of computing routing tables.

15.1 Data Members

15.1.1 The `_id`

unsigned int the_id;

The id of a node.

15.1.2 The `_cost`

unsigned int the_cost;

The cost of a packet reaching this node.

15.1.3 The `_parent_branch`

unsigned int the_parent_branch;

The branch toward the root node that this node is connected to.

15.2 Functions

Public Functions

15.2.1 `Vertex()`

Vertex(int i, int c, int p)

The constructor of Vertex has the following inputs:

- i:** Unique id number of the Vertex node.
- c:** Cost associated with the Vertex.
- p:** Parent branch of the Vertex.

The corresponding data members are initialised to equal the above variables.

15.2.2 `Id()`

unsigned int id(void)

Returns the value of the `_id`.

15.2.3 `Cost()`

unsigned int cost(void)

Returns the value of the `_cost`.

15.2.4 Parent_branch()

unsigned int parent_branch(void)

Returns the value of the `_parent_branch`.

15.2.5 Operator

bool operator>(Vertex& b)

bool operator<(Vertex& b)

bool operator==(Vertex& b)

These three functions overload the operators `>`, `<` and `==`. They operate according to the `_cost` of each node.

16 SFC_Router

An SFC Router is an extension of the standard Router (see Section 14) that supports the SFC protocol (TCP/RAMI) developed by CUBIN (University of Melbourne). In the real world, an SFC Router would be situated at the interface between the access network and the Internet. Operates much like a Router, with the exception being that a value (a function of the current queue size) is inserted into every packet's payload at the time of transmission.

16.1 #Defines

16.1.1 SFC_ROUTER_SERVICE_INTERVAL

The time interval that the SFC Router periodically services itself (i.e. how often `service()` is executed).

16.1.2 SFC_ROUTER_QUEUE_AVERAGE_SAMPLES

Specifies the number of queue size samples taken in one service interval.

16.1.3 SFC_ROUTER_ESTIMATOR_SMOOTH

The maximum number of packet departure times stored for the SFC Router.

16.1.4 SFC_INIT_RATE_EST

The initial transmission (channel) rate estimate of the SFC Router.

16.2 Structures

16.2.1 SFC_Payload

The specification of the header for a packet that passes through an SFC Router is included below:

```

typedef struct {
    double          pq_func;          // P(q) function calculated by algorithm
    unsigned int    old_protocol;     // protocol of original packet
    void            *rest_payload;    // payload of original packet
} SFC_Payload;

```

16.3 Data Members

16.3.1 Max_line_rate

unsigned int max_line_rate;

The maximum line rate of the SFC Router. Depends on the constructor input parameter “mr”.

16.3.2 SFC_q_min

double SFC_q_min;

A user defined parameter involved in the calculation of the P(q) function.

16.3.3 SFC_b_param

double SFC_b_param;

A user defined parameter involved in the calculation of the P(q) function.

16.3.4 SFC_chan_rate

double SFC_chan_rate;

An estimate of the channel (output) rate of the SFC Router.

16.3.5 Pac_deps

deque<pair<double, int> > pac_deps;

A container that stores the departure times (and size) of packets leaving the SFC Router. Also used to calculate an estimate for the output rate.

Key (double): Time of packet’s departure.

Data (int): Size of the packet.

16.3.6 Queue_samples

deque<unsigned int> queue_samples;

Stores the queue size over the last service intervals. Is also used to calculate average queue size.

16.3.7 Queue_acc

unsigned int queue_acc;

Utilised in the calculation of mean queue size.

16.3.8 Mean_queue

double mean_queue;

The average queue size of an SFC Router. Also used to calculate P(q) function.

16.3.9 Data_out

unsigned int data_out;

The amount of data that has been output (i.e. the total size of all packets output from the SFC Router).

16.4 Functions

Note: An SFC Router utilises the Router's in() function when PASS events (see Section 18.1.1) are processed:

Public Functions

16.4.1 SFC_Router()

SFC_Router(int i, unsigned int bl, double q_min, double b_param, unsigned int mr)

The constructor of SFC_Router has the following input parameters:

- i:** Unique id number of SFC Router node.
- bl:** Maximum size of an output queue for this SFC Router.
- q_min:** Parameter involved in the calculation of the P(q) function.
- b_param:** Parameter involved in the calculation of the P(q) function.
- mr:** Maximum output line rate.

A Router (see Section 14) object is initialised with the same id number and buffer limit as the SFC Router. The constructor itself initialises most of the data members then schedules a SERVICE event (see Section 18.1.1) that will periodically calculate average queue size (see Section 16.4.4).

16.4.2 Out_event()

Packet out_event(Event *the_event)*

The input parameter "the_event" contains data that specifies which port a packet exits the SFC Router from. Initially, update_estimators() (see Section 16.4.1) is called to maintain pac_deps (see Section 16.3.5) and also to estimate the SFC_chan_rate (see Section 16.3.4). Each packet is transformed into an SFC packet (that encapsulates the original packet) prior to being output and the current P(q) value is evaluated (by calling calc_pq() - see Section 16.4.2) and stored inside it.

16.4.3 Log()

void log(void)

Various statistics are printed to standard output for analysis. This function uses an output string stream (ostream) object to hold the statistics that are printed.

16.4.4 Service()

void service(void)

The average queue size (see Section 16.3.8) of the SFC Router is calculated by this function. The average queue size is a parameter used in the calculation of the P(q) function.

Private Functions

16.4.1 Update_estimators()

void update_estimators(unsigned int p_size)

A function which maintains various statistics of an SFC Router whenever a packet leaves. These statistics are involved in the calculation of the P(q) function.

16.4.2 Calc_pq()

double calc_pq(void)

This function uses the values of mean_queue and SFC_chan_rate, as well as the user defined parameters SFC_q_min and SFC_b_param to calculate a linear value of P(q) for the SFC Router at the current point in time. This value is stored in the header of the next packet that leaves the SFC Router.

17 SFC_Rec

An SFC_Rec (or SFC Filter) object acts as a filter between a link and a receiver. Information stored by a SFC Router (see Section 16) in the headers of SFC packets are used to update state variables such as the window size. Other types of packets are simply forwarded to the receiver (Router).

17.1 #Defines

17.1.1 SFC_REC_SMOOTH_FACTOR

The maximum number of packet arrival times recorded.

17.1.2 SFC_REC_INIT_W_SIZE

An initial estimate of the window size.

17.1.3 SFC_REC_BUFFER

The buffer limit of the receiver attached to the filter.

17.1.4 SFC_REC_RATE_TIME_EST

Not implemented by SFC_Rec.

17.2 Data Members

17.2.1 Awnd_size

double awnd_size;

The current advertised window size.

17.2.2 Last_tcp_awnd

unsigned int last_tcp_awnd;

Advertised window size of receiver, obtained from the last ack received.

17.2.3 Pac_recs

deque<pair<double, unsigned int> > pac_recs;

This container stores the arrival times (and size) of packets entering the SFC Filter. It is also used to calculate an estimate for the input rate.

Key (double): Time of packet arrival.

Data (unsigned int): Size of the packet.

17.2.4 Cur_rate

double cur_rate;

An estimate of the current input channel rate of the SFC Filter.

17.2.5 Data_in

unsigned int data_in;

The total amount of data received at the SFC Filter (the total size of all arriving packets).

17.2.6 Sfc_max_delta

unsigned int sfc_max_delta;

A user defined variable involved in the calculation of `awnd_size` (advertised window size).

17.2.7 Sfc_tau

unsigned int sfc_tau;

A user defined parameter that is indirectly involved in the calculation of `awnd_size`.

17.2.8 Last_pq

double last_pq;

The value of the $P(q)$ function contained in the last packet that arrived at the SFC Filter.

17.2.9 Rtt

double rtt;

The round trip time variable is used to calculate a temporary value which is then used to calculate `awnd_size`.

17.2.10 Last_ack_num

unsigned int last_ack_num;

The sequence number of the last ack received determines the number of duplicate acks in the prime start algorithm.

17.2.11 Dup_acks

unsigned int dup_acks;

This variable (the number of duplicate acks) is used in the prime start algorithm.

17.2.12 Prime_start

bool prime_start;

Signifies whether the prime start algorithm has been activated.

17.3 Functions

Public Functions

17.3.1 SFC_Rec()

SFC_Rec(const int id, unsigned int smd, unsigned int sfct)

The constructor of `SFC_Rec` has the following inputs:

- id:** Unique id number of SFC Rec node.
- smd:** Involved in calculation of `awnd_size`.
- sfct:** Involved in calculation of `awnd_size`.

The Router (see Section 14) object within the `SFC_Rec` object is initialised with the same id number as the SFC Filter and a buffer limit equal to `SFC_REC_BUFFER`

(which is in effect, infinite) by way of the initialisation list in the constructor. Within the constructor, the various state variables are initialised, as are the variables associated with the prime start algorithm (this only occurs if the line "#define PRIME_START" is found at the top of the source file).

17.3.2 In()

*void in(Packet *p)*

Processes packets arriving at the SFC Filter. Data packets with TCP (see Section 22.1.2) (or any other protocol) in their header are simply forwarded (without alteration) to the receiver (Router). Data packets with SFC (see Section 22.1.2) in their protocol field are stripped of the information in their headers (to update various state variables by calling `update_estimators()` (see Section 17.3.2)) and then restored to their original form and forwarded to the receiver. If the packet is a TCP acknowledgement (ack) packet, the `last_tcp_awnd` variable is updated. The prime start algorithm is only executed if the line "#define PRIME_START" is at the top of the source file.

17.3.3 Log()

void log(void)

Various statistics are appended to an output string stream (`ostringstream`) object, whose contents are then printed to standard output for observation.

17.3.4 Service()

void service(void)

This function is not defined as periodic servicing of an SFC Filter is not required.

Private Functions

17.3.1 Forward_packet()

*void forward_packet(Packet *p)*

This function passes a packet to the receiver attached to the SFC Filter. It calls the `in()` (see Section 14.3.2) function defined in the Router class.

17.3.2 Update_estimators()

void update_estimators(unsigned int p_size, double pq)

Called when a packet of type SFC (see Section 22.1.2) arrives at the filter. It uses the information removed from the packet's header to update and calculate various state variables. Initially the container `pac_recs` is updated with the arrival time and size of the packet, and then an estimate of the input channel rate of the filter is calculated. The rest of the function deals with the computation of `awnd_size`. The size of the advertised window must also be controlled in case of bottlenecks occurring in the core network.

17.3.3 Get_next_packet()

Packet get_next_packet(void)*

Function not defined for SFC_Rec.

17.3.4 Min()

inline double min(double x, double y)

Returns whichever is the minimum value of x and y.

17.3.5 Max()

inline double max(double x, double y)

Returns whichever is the maximum value of x and y.

18 Event

Instances of this class represent the various types of events that occur in a simulation. When these events occur, event objects are placed into the Event_list (see Section 20) and serviced at a later time. Servicing one event object may also cause the creation of a new event object (also placed in the Event_list). The type of event is specified by the value of the EVENT_TYPE argument in the constructor.

18.1 Structures

18.1.1 EVENT_TYPE

An enumerated type that specifies the particular kind of event that can occur in a simulation. A brief description of each event is also included:

```
typedef enum EVENT_TYPE {  
    PASS,  
    START,  
    STOP,  
    LOG,  
    SERVICE  
};
```

PASS: This event represents a packet that moves from one node to another on its journey through the network. It causes the execution of the sending node's out() (see Section 2.2.4) function and the receiving node's in() (see Section 2.2.2) function.

START: This signifies that a node begins operating. It causes the execution of the node's start() (see Section 2.2.6) function.

STOP: This event signifies that a node deactivates itself and stops sending or receiving packets. It causes the execution of the node's `stop()` (see Section 2.2.7) function.

LOG: This event represents a log entry. A log entry's purpose is to provide the user with statistics and diagnostic information, such as packet arrival times at a TCP node (see Section 13.4.4). It causes the execution of the node's `log()` (see Section 2.2.8) function.

SERVICE: This event performs a servicing function for a node at a regular interval such as checking for time-outs (see Section 13.4.5) and calculating mean queue size (see Section 16.4.4). It causes the execution of the node's `service()` (see Section 2.2.9) function.

18.2 Data Members

18.2.1 Time

double time;

This variable represents the time for which the event will occur (and the time that it will be serviced).

18.2.2 Repeat_time

double repeat_time;

This variable specifies the interval that periodic events will occur at. After a periodic event is handled, the `repeat_time` is added to the current simulation time and placed back into the `Event_list` (see Section 20) (the same event is serviced again after the duration of `repeat_time` has elapsed).

18.2.3 Node

int node;

The id of the node that scheduled the event.

18.2.4 Type

EVENT_TYPE type;

Represents the type of event. The type depends on the value of the enumerated variable `EVENT_TYPE`.

18.2.5 The_data

*void *the_data;*

A pointer to the data in the event object. `The_data` holds extra information about the event, e.g. the port number (of a router) that a packet is output from.

18.3 Functions

Public Functions

18.3.1 Event()

The constructor is overloaded (there are three different constructors). The constructors have some or all of the following 5 inputs:

- t:** Time that event occurs.
- n:** Id of the node that scheduled the event.
- ty:** Type of event (either PASS, START, STOP, LOG or SERVICE).
- rt:** Repeat time. Has non-zero value if event is repetitive.
- *d:** Pointer to the data associated with the particular event.
e.g. output port number of the router that a packet is output from.

The three different constructors for an event object are:

Event(double t, unsigned int n, EVENT_TYPE ty, double rt)

A general periodic event is initialised by this constructor.

Event(double t, unsigned int n, EVENT_TYPE ty)

A single event with no data attached is initialised by this constructor.

*Event(double t, unsigned int n, EVENT_TYPE ty, void *d)*

A single event with data attached is initialised by this constructor.

Useful for nodes which need more information than the general event.

18.3.2 Execute()

bool execute(void)

The execute() function acts as the event handler for each event object in the Event_list (see Section 20). It basically delegates which functions the nodes will need to execute to service the event, depending on the value of the EVENT_TYPE field. It also determines whether events will be re-inserted into the Event_list (periodic events).

18.3.3 Get_time()

double get_time(void)

This function returns the value of the variable time, which is when the event was scheduled for.

18.3.4 Get_node()

int get_node(void)

This function returns the value of the integer node, which is the id of the node that scheduled the event.

18.3.5 Get_data()

void get_data(void)*

Returns a pointer to the data associated with the particular event.

18.3.6 Operator

bool operator<(Event& b)

bool operator>(Event& b)

bool operator==(Event& b)

The three operators <, > and == have been overloaded to deal with event objects. They compare the time of the current event object with that of another event object and return the corresponding boolean value.

19 E_comp

This is a very small class that is used as the comparator for the priority queue container in the Event_list class (see Section 20).

19.1 Functions

Public Functions

19.1.1 Operator()

*bool operator() (Event *a, Event *b)*

The operator () has been overloaded to handle event objects. It basically returns true if event a's time is greater than event b's, and false otherwise.

20 Event_list

This class consists of a priority queue, which is the queue that events are inserted into when they are scheduled. There is only one instance of this class in the simulator, and it is therefore declared global to allow all nodes to schedule events.

20.1 Data Members

20.1.1 Event_q

priority_queue<Event, vector<Event*>, e_comp > event_q;*

This is the container which holds a pointer to each event object inserted into the event list. This queue is traversed by the main module which services each event separately. The field e_comp (see Section 19) is the comparator of the container. It ensures that the first element is the most recent event, and the last element (which will be serviced next) is the oldest.

20.1.2 Cur_time;

double cur_time;

This maintains the current time of the simulation. Events are scheduled with reference to this time.

20.2 Functions

Public Functions

20.2.1 Event_list()

Event_list(void)

The constructor initialises cur_time to zero.

20.2.2 Insert()

*void insert(Event *e)*

This function inserts an event into the event_q container.

20.2.3 Get_next()

Event get_next(void)*

Removes the event which is to be serviced next.

20.2.4 Get_cur_time()

double get_cur_time(void)

Returns the value of cur_time.

20.2.5 Size()

int size(void)

Returns the current size of the event_q container (the priority queue of events).

21 Main

This is the heart of the simulation. The bulk of the simulation is conducted in the main module. The main module parses each line of the World File and constructs the corresponding network object. Once the entire network has been constructed, all the nodes in the network are initialised (by calling their initialise() (see Section 14.3.8) function) and the event queue is executed (serviced).

21.1 #Defines

21.1.1 MAX_ARGS

This value is set arbitrarily higher than the maximum number of arguments that a network object can take. It is the size of the array which is used to store the arguments of the particular object.

21.1.2 LINK_ARGS

Maximum number of arguments a node of type Link can take.

21.1.3 SLINK_ARGS

Maximum number of arguments a node of type SLink can take.

21.1.4 GEN_ARGS

Maximum number of arguments a node of type Generator can take.

21.1.5 TCP_ARGS

Maximum number of arguments a node of type TCP can take.

21.1.6 ROUTER_ARGS

Maximum number of arguments a node of type Router can take.

21.1.7 SFC_ROUTER_ARGS

Maximum number of arguments a node of type SFC_Router can take.

21.1.8 SFC_FILTER_ARGS

Maximum number of arguments a node of type SFC_Rec can take.

21.1.9 TS_GREEDY

Maximum number of arguments a node of type TS_Greedy can take.

21.1.10 TS_EXP

Maximum number of arguments a node of type TS_Exp can take.

21.1.11 TP_TWO_STATE

Maximum number of arguments a node of type TP_two_state can take.

21.1.12 DELIM_TOKENS

String used for parsing the World File. These characters are insignificant and any other characters should be part of valid information, unless the information is commented out.

21.1.13 MAX_LINE

Defined but not used in the simulator.

21.2 Global Variables

21.2.1 Event_list

Event_list event_list;

This instance of the class `Event_list` (see Section 20) represents the priority queue which contains all the events that have been scheduled. It is defined global because nearly all network objects insert events into the queue.

21.2.2 Node_map

Node_map node_map;

This is an instance of the class `Node_map` (see Section 24). It maintains a list of all the nodes' ids and their pointers to memory.

21.2.3 Con_table

Con_table con_table;

This is an instance of the class `Con_table` (see Section 25), which keeps track of the interconnections between nodes in the network.

21.3 Functions

Public Functions

21.3.1 Parse()

*void parse(istream& f, double *end_time)*

This function parses the first word of each line in the World File and then calls the corresponding function to create the object defined on that line. In the definitions below, the words in brackets can be found in a World File). There are five main words that indicate the definition of an object. They are:

source: Indicates that the object is a Traffic Source (see Section 5). There are two types of Traffic Source: Greedy (see Section 7) (`ts_greedy`) or Exponential (see Section 6) (`ts_exp`). The function `add_source()` is called to construct the Traffic Source object.

process: Signifies that the object is a Transmit Process (see Section 8). There are two types of Transmit Process: general (see Section 10) (`tp_general`) and two state (see Section 9) (`tp_two_state`). The function `add_process()` is executed to create the Transmit Process object.

node: Specifies that the object is a Node (see Section 2). There are several types of node (not including Traffic Source): Link (see Section 3) (`link`), Stochastic Link (see Section 4) (`slink`), standard Router (see Section 14) (`router`), SFC Router (see Section 16) (`sfc_router`), SFC Filter (see Section 17) (`sfc_rec`), Generator (see Section 12) (`generator`) and TCP (see Section 13) (`tcp`). The function `build_node()` is executed to construct the Node object.

connect: Denotes that the object defined is a `Con_table` (see Section 25) entry (a connection between two nodes). The function `add_connection()` is called to add the entry to the table.

event: Indicates that the object is an Event (see Section 18). There are three types of events that can be defined in the world file: START (see Section 18.1.1) (`start`), STOP (see Section 18.1.1) (`stop`) and LOG (see Section 18.1.1) (`log`). The function `add_event()` is executed to create the event object.

21.3.2 Add_source()

void add_source(const string& line, string::size_type p)

Creates a Traffic Source object with the arguments given in the World File. The Traffic Source can be either Greedy or Exponential.

21.3.3 Add_process()

void add_process(const string& line, string::size_type p)

Creates a Transmit Process object with the arguments provided in the World File. The Transmit Process can be either `two_state` or `general`.

21.3.4 Build_node()

void build_node(const string& line, string::size_type p)

Creates a Node object with the arguments included in the World File. The Node can be either a Link, SLink, Generator, TCP node, Router, SFC Router or SFC Filter (`SFC_Rec`).

21.3.5 Add_connection()

void add_connection(const string& line, string::size_type p)

Creates a new entry in the `Con_table`. The new entry represents a connection between two nodes in the network.

21.3.6 Add_event()

void add_event(const string& line, string::size_type p)

Creates a new event object of either type START, STOP or LOG and places it into the event queue. These are the only events that must be defined and created before the simulation begins.

21.3.7 Get_next_word()

*const string get_next_word(const string& s, string::size_type *p)*

Parses the next word from a file. This function is used extensively when parsing the world file.

21.3.8 Out_of_store()

void out_of_store(void)

Prints an error message when the computer/program is out of memory.

22 Packet

A packet is the basic unit of information that is distributed throughout a network. The packet modelled by the QNS complies with the Internet Protocol (IP) specification. This means that each packet has its own IP header with information such as source and destination addresses and various other fields.

22.1 Structures

22.1.1 IPHeader

The packet structure conforms with the IP version 4 format by containing all the necessary fields. The following is the structure for a packet header:

```

typedef      struct{
              unsigned int  version;
              unsigned int  header_length;
              unsigned int  TOS;
              unsigned int  total_length;
              unsigned int  ident;
              bool          flag1, flag2, flag3;
              unsigned int  frag_offset;
              unsigned int  TTL;
              unsigned int  protocol;
              unsigned int  checksum;
              unsigned int  src;                // source address.
              unsigned int  dst;               // destination address.
              unsigned int  option1, option2, option3, option4; // options - if any
            } IPHeader;

```

22.1.2 PROTOCOL

The following enumerated type defines the type of protocol being used. It also indicates the source of each packet.

```

typedef enum  PROTOCOL {
              RAW,                // Packet is from a Traffic Source
              GENERATOR,         // Packet is from a Generator node.
              TCP_PROTOCOL,     // Packet is from a TCP node.
              UDP,               // Not used.
              SFC                 // Packet is from an SFC Router.
            };

```

22.2 Data Members

22.2.1 The_header

IPHeader the_header;

This is the header of the packet. The structure for IPHeader is defined above.

22.2.2 Pac_size

unsigned int pac_size;

The size of the packet in bytes.

22.2.3 Payload

*void *payload;*

A pointer to the payload of the packet. Payloads are created by the various nodes (TCP, Generator, SFC Router).

22.3 Functions

Public Functions

22.3.1 Packet()

Packet(int s)

The constructor for a packet takes as input the packet's size, "s", in bytes and initialises `pac_size` with it's value.

22.3.2 Size()

int size(void)

Returns the integer `pac_size` (the packet's size).

22.3.3 Header()

IPHeader header(void)*

Returns a pointer to the header of the packet.

22.3.4 Set_payload

*void set_payload(void *p)*

Attaches the payload, pointed to by "p", to the packet.

22.3.5 Get_payload()

*void *get_payload(void)*

Returns a pointer to the payload of the packet.

22.3.6 Destination()

unsigned int destination(void)

Returns the id of the destination node of the packet.

22.3.7 Source()

unsigned int source(void)

Returns the id of the source node of the packet.

22.3.8 Protocol()

unsigned int protocol(void)

Returns the protocol of the packet.

23 Routes

This module consists of two classes: `Node_map` and `Con_table`. This module keeps track of all the node objects, and also maintains a table of connections between them.

24 Node_map

This class stores pointers to the various node objects in memory. Many classes (modules) in the simulator manipulate an instance of this class and it is therefore defined global.

24.1 Data Members

24.1.1 Nodes

```
map<int, Node*> nodes;
```

This is a map of pointers to all nodes in the network.

Key (int): Id of the node.

Data (Node*): Pointer to the area of memory where the node is located.

24.2 Functions

Public Functions

24.2.1 Node_map()

```
Node_map()
```

The constructor does not perform any initialisations.

24.2.2 Add_node()

```
void add_node(Node *the_node, const int the_id)
```

This function adds the id of a node and its corresponding pointer into the map container called nodes.

24.2.3 Id2node()

```
Node* id2node(const int)
```

This function, when specified a particular node's id, will return a pointer to it.

24.2.4 Exists()

```
bool exists(const int i)
```

This function returns TRUE if the node with id 'i' exists in the nodes container.

24.2.5 Num_nodes()

unsigned int num_nodes(void)

Returns the number of nodes in the network topology.

24.2.6 Get_all_nodes()

vector<int> get_all_nodes(void)*

This function returns a vector containing all nodes in the network. It is imperative that the function that calls `get_all_nodes()` disposes of the vector when it has finished using it.

25 Con_table

This class stores the "connection table" that defines the connections between nodes and the corresponding costs to get from one to the other. This class relies on the boost graph library from: www.boost.org

25.1 Data Members

25.1.1 Connection

multimap<int, pair<int, int> > connection;

This multimap container stores the connections between nodes the network. The node where the connection begins is referred to as the source and the destination node is where it ends.

Key (int): The source node's id.

Data (first int): The destination node's id.

Data (second int): The cost from source to destination.

25.2 Functions

Public Functions

25.2.1 Con_table()

Con_table()

The constructor, doesn't perform any initialisations.

25.2.2 Add_con()

void add_con(int src, int dst)

This function adds a new connection entry between a node with id "src" and a node with id "dst". The entry in the connection multimap is given a default cost of 1.

void add_con(int src, int dst, cost c)

Same as above but this type of entry in the connection multimap is given the cost of “c”.

25.2.3 Is_connected()

bool is_connected(const int src, const int dst)

Returns true if node with id “src” is connected to the node with id “dst”.

25.2.4 Num_outputs()

unsigned int num_outputs(int src)

Returns the number of outputs of a particular node.

25.2.5 Get_output()

unsigned int get_output(int src)

Returns the first output a node with id “src” is connected to.

25.2.6 Get_outputs()

vector<pair<int,int> > get_outputs(const int src)*

This function returns all the outputs for a node with id “src”. It is imperative that the function that calls get_outputs() disposes of the vector once it has finished using it.

26 Constants

This class contains some defined constants used in many of the classes (modules) in the simulator.

26.1 #Defines

26.1.1 BITS_PER_BYTE

Number of bits per byte.

26.1.2 DEFAULT_TTL

The default value for the field TTL (time to live) in the IP header.

26.1.3 ACK_SIZE

The default size for an acknowledgement.

26.1.4 MAX_DELAY

This value is chosen to be extremely large, simulating an infinitely long delay.

27 ICMP

The ICMP protocol is used to report problems with delivery of IP datagrams (packets) within an IP network. Situations where it is used include: to indicate that a particular destination system is not responding, to signify that an IP network is not reachable, to signal that a node is overloaded, to specify that an error occurred in the IP header information, etc. ICMP packets are encapsulated in IP packets for transmission across a network.

27.1 Structures

27.1.1 ICMP_type

```
typedef enum{
    ECHO_REPLY,
    ECHO_REQUEST,
    NET_UNREACHABLE,
    HOST_UNREACHABLE,
    PROTOCOL_UNREACHABLE,
    PORT_UNREACHABLE,
    CAN_NOT_FRAG,
    SOURCE_ROUTE_FAIL,
    DEST_NET_UNKNOWN,
    SOURCE_QUENCH,
    ROUTER_ADV,
    ROUTER_SOL,
    TTL_ZERO,
    IP_HEADER_BAD
} ICMP_type;
```

This structure doesn't contain all the ICMP control messages, but contains all the major ones necessary to facilitate the simulation.

27.2 Functions

27.2.1 Send_ICMP_message()

void send_ICMP_message(unsigned int dest, ICMP_type t)

Create an ICMP packet with a message and send it. This function has not been implemented.

Index

- #define
 - Constants
 - ACK_SIZE, 66
 - BITS_PER_BYTE, 66
 - DEFAULT_TTL, 66
 - MAX_DELAY, 66
 - Main
 - DELIM_TOKENS, 59
 - GEN_ARGS, 58
 - LINK_ARGS, 58
 - MAX_ARGS, 58
 - MAX_LINE, 59
 - ROUTER_ARGS, 58
 - SFC_FILTER_ARGS, 58
 - SFC_ROUTER_ARGS, 58
 - SLINK_ARGS, 58
 - TCP_ARGS, 58
 - TP_TWO_STATE, 58
 - TS_EXP, 58
 - TS_GREEDY, 58
 - Router
 - ROUTER_ESTIMATOR_SMOOTH, 41
 - ROUTER_SERVICE_INTERVAL, 41
 - SFC_Rec
 - SFC_REC_BUFFER, 50
 - SFC_REC_INIT_W_SIZE, 49
 - SFC_REC_RATE_TIME_EST, 50
 - SFC_REC_SMOOTH_FACTOR, 49
 - SFC_Router
 - SFC_INIT_RATE_EST, 46
 - SFC_ROUTER_ESTIMATOR_SMOOTH, 46
 - SFC_ROUTER_QUEUE_AVERAGE_SAMPLES, 46
 - SFC_ROUTER_SERVICE_INTERVAL, 46
 - TCP
 - DEFAULT_AWND_SIZE, 35
 - TCP_INIT_SS_THRESH, 35
 - TCP_INITIAL_RTO, 35
 - TCP_MAX_DUP_ACKS, 35
 - TCP_RTO_MAX, 35
 - TCP_RTO_MIN, 35
 - TCP_SERVICE_INTERVAL, 35
- abstract, 21, 25, 31
- abstract class, 21, 25, 31
- acknowledgement (ack), 32, 33, 38, 52
- advertised window size, 50, 52
- bandwidth, 17, 18, 20
- buffer limit, 42, 48
- Con_table, 44, 60, 65
 - add_con(), 65
 - Con_table(), 65
 - connection, 65
 - get_output(), 66
 - get_outputs(), 66
 - is_connected(), 66
 - num_outputs(), 66
- congestion window, 39
- Constants, 66
 - ACK_SIZE, 66
 - BITS_PER_BYTE, 66
 - DEFAULT_TTL, 66
 - MAX_DELAY, 15, 66
- Constructors
 - Con_table(), 65
 - Event()
 - Periodic event, 55
 - Single event, 55
 - Single event with data attached, 55
 - Event_list(), 57
 - Gen_state(), 30
 - Generator(), 32
 - Link(), 17
 - Node(), 15

- Node_map(), 64
- Packet(), 63
- Router(), 42
- SFC_Rec(), 51
- SFC_Router(), 48
- SLink(), 20
- TCP(), 38
- TP_general(), 29
- TP_two_state(), 27
- Traffic_source(), 21
- TS_Exp(), 23
- TS_Greedy(), 24
- Vertex(), 45
- Container
 - Con_table
 - connection, 65
 - Event_list
 - event_q, 56
 - Gen_state
 - cum_prob, 30
 - trans_time, 30
 - Generator
 - buff, 31
 - Link
 - buff, 17
 - Node_map
 - nodes, 64
 - Router
 - buff, 41
 - drop_table, 42
 - routing_table, 42
 - usage_table, 42
 - SFC_Rec
 - pac_recs, 50
 - SFC_Router
 - pac_deps, 47
 - queue_samples, 47
 - TCP
 - pac_rcv, 38
 - pac_snt, 37
 - TP_general
 - states, 28
 - Traffic_source
 - buff, 21
- cumulative mass function (cmf), 28–30
- cumulative probability, 28
- delay, 17, 20
- Dijkstra's Algorithm, 44
- e_comp, 56
 - operator(), 56
- Enumerated Type
 - Event
 - EVENT_TYPE, 53
 - Packet
 - PROTOCOL, 62
 - TP_two_state
 - CHAN_STATE, 26
- Event, 53
 - Event(), 55
 - EVENT_TYPE, 53
 - execute(), 55
 - get_data(), 56
 - get_node(), 55
 - get_time(), 55
 - node, 54
 - operator<(), 56
 - operator==(), 56
 - operator>(), 56
 - repeat_time, 54
 - the_data, 54
 - time, 54
 - type, 54
- event handler, 18, 55
- Event types
 - LOG, 16, 54, 60, 61
 - PASS, 15, 23, 25, 32, 33, 39, 43, 48, 53
 - SERVICE, 16, 33, 39, 42, 48, 54
 - START, 53, 60, 61
 - STOP, 54, 60, 61
- Event_list, 53, 56
 - cur_time, 57
 - Event_list(), 57
 - event_q, 56
 - get_cur_time(), 57
 - get_next(), 57
 - insert(), 57
 - size(), 57

- fast recovery/retransmit mode, 40
- flow control, 31, 35
- Gen_state, 29
 - add(), 30
 - cum_prob, 30
 - Gen_state(), 30
 - get_prob(), 30
 - get_sojourn_time(), 30
 - get_time(), 31
 - last_prob(), 30
 - set_sojourn_time(), 30
 - sojourn_time, 29
 - trans_time, 30
- Generator, 17, 21, 23–25, 31, 32, 62
 - ack_received(), 34
 - buff, 31
 - dest, 32
 - dest_node(), 33
 - Gen_payload, 31
 - Generator(), 32
 - get_new_packet(), 34, 40
 - get_next_packet(), 34, 39
 - get_w_size(), 34
 - in(), 32
 - in_flight, 32
 - in_flight(), 34
 - last_received, 32
 - last_sent, 32
 - log(), 33
 - out(), 33
 - queue_packet(), 34
 - service(), 33
 - set_in_flight(), 34
 - set_w_size(), 34
 - start(), 33
 - ts, 31
 - w_size, 31
- Global Variables, 59
 - Con_table, 59
 - Event_list, 59
 - Node_map, 59
- ICMP, 67
 - ICMP_type, 67
 - Send_ICMP_message(), 67
- initialisation list, 17, 20, 22–24, 42, 52
- Internet Protocol (IP) specification, 61
- Link, 17, 20
 - bandwidth(), 18
 - buff, 17
 - busy(), 18
 - delay(), 19
 - get_next_packet(), 19
 - in(), 18
 - in_flight(), 19
 - Link(), 17
 - log(), 19
 - out(), 18
 - prop_delay, 17
 - queue_packet(), 19
 - rate, 17
 - serve(), 19
 - service(), 19
 - set_time_to_free(), 19
 - start(), 18
 - stop(), 18
 - time_to_free, 17
 - when_can_send(), 18
- loss probability, 20
- loss rate, 20
- lossless transmission, 26, 27
- Main, 57
 - add_connection(), 60
 - add_event(), 61
 - add_process(), 60
 - add_source(), 60
 - build_node(), 60
 - con_table, 59
 - DELIM_TOKENS, 59
 - event_list, 59
 - GEN_ARGS, 58
 - get_next_word(), 61
 - LINK_ARGS, 58
 - MAX_ARGS, 58
 - MAX_LINE, 59
 - node_map, 59
 - out_of_store(), 61

- parse(), 59
- ROUTER_ARGS, 58
- SFC_FILTER_ARGS, 58
- SFC_ROUTER_ARGS, 58
- SLINK_ARGS, 58
- TCP_ARGS, 58
- TP_TWO_STATE, 58
- TS_EXP, 58
- TS_GREEDY, 58
- maximum segment size, 38
- mean separation time, 23
- network object, 57, 59
- Node, 14, 31
 - busy(), 15
 - id(), 16
 - in(), 15, 21
 - initialise(), 16
 - log(), 21
 - next_dest(), 16
 - Node(), 15
 - out(), 15, 21
 - out_event(), 15
 - running(), 16
 - service(), 16, 21
 - start(), 15, 39
 - stop(), 16
 - stopped, 14
 - the_id, 14
- Node_map, 64
 - add_node(), 64
 - exists(), 64
 - get_all_nodes(), 65
 - id2node(), 64
 - Node_map(), 64
 - nodes, 64
 - num_nodes(), 65
- output string stream (ostringstream),
 - 19, 21, 33, 41, 43, 49, 52
- P(q) function, 47–49, 51
- Packet, 61
 - destination(), 63
 - get_payload(), 63
 - header(), 63
 - IPHeader, 61
 - pac_size, 62
 - Packet(), 63
 - payload, 62
 - PROTOCOL, 62
 - protocol(), 63
 - set_payload(), 63
 - size(), 63
 - source(), 63
 - the_header, 62
- payload, 33, 38, 46, 47, 62, 63
- prime start algorithm, 51, 52
- propagation delay, 17, 20
- protocol, 62, 63
 - GENERATOR, 32, 33
 - RAW, 32, 38
 - SFC, 52
 - TCP, 38, 52
- purely virtual, 14, 21, 22
- purely virtual functions
 - Node
 - in(), 15
 - log(), 16
 - out(), 15
 - service(), 16
 - Traffic_source
 - get(), 22
 - lost(), 25
 - time(), 25
 - transmit(), 25
- queue, 17
- rate, 17, 18
- Retransmission Time Out (RTO), 35, 39–41
- Round Trip Time (RTT), 40
- Router, 16, 17, 41, 46, 51
 - buff, 41
 - buffer_limit, 41
 - drop_table, 42
 - get_next_packet(), 44
 - in(), 43, 48
 - initialise(), 43
 - log(), 43
 - mem_used(), 44

- next_dest(), 43
- next_hop(), 44
- out(), 43
- out_event(), 43
- packets_dropped(), 44
- queue_packet(), 44
- queue_size(), 44
- Router(), 42
- ROUTER_ESTIMATOR_SMOOTH, 41
- ROUTER_SERVICE_INTERVAL, 41
- routing_table, 42
- service(), 43
- update_routing_table(), 44
- usage_table, 42
- Routes, 64
- Routing Table, 16, 43–45
- sequence number, 32, 40
- SFC Filter, 49, 51
- SFC packet, 48, 49, 52
- SFC Router, 46, 49, 62
- SFC_Rec, 49
 - awnd_size, 50
 - cur_rate, 50
 - data_in, 50
 - dup_acks, 51
 - forward_packet(), 52
 - get_next_packet(), 53
 - in(), 52
 - last_ack_num, 51
 - last_pq, 51
 - last_tcp_awnd, 50
 - log(), 52
 - max(), 53
 - min(), 53
 - pac_recs, 50
 - prime_start, 51
 - rtt, 51
 - service(), 52
 - sfc_max_delta, 50
 - SFC_Rec(), 51
 - SFC_REC_BUFFER, 50
 - SFC_REC_INIT_W_SIZE, 49
 - SFC_REC_RATE_TIME_EST, 50
 - SFC_REC_SMOOTH_FACTOR, 49
 - sfc_tau, 51
 - update_estimators(), 52
- SFC_Router, 46
 - calc_pq(), 49
 - data_out, 48
 - log(), 49
 - max_line_rate, 47
 - mean_queue, 48
 - out_event(), 48
 - pac_deps, 47
 - queue_acc, 47
 - queue_samples, 47
 - service(), 49
 - SFC_b_param, 47
 - SFC_chan_rate, 47
 - SFC_INIT_RATE_EST, 46
 - SFC_Payload, 46
 - SFC_q_min, 47
 - SFC_Router(), 48
 - SFC_ROUTER_ESTIMATOR_SMOOTH, 46
 - SFC_ROUTER_QUEUE_AVERAGE_SAMPLES, 46
 - SFC_ROUTER_SERVICE_INTERVAL, 46
 - update_estimators(), 49
- SLink, 20
 - cur_xmit_time, 20
 - in(), 20
 - log(), 21
 - service(), 21
 - SLink(), 20
 - trans_func, 20, 21
- slow start threshold, 35
- state description, 28
- State Specification File, 28, 29
- state variables, 21, 33, 36–38, 40, 52
- Stochastic Flow Control (SFC), 46
- Structure
 - Generator
 - Gen_payload, 31
 - ICMP

- ICMP_type, 67
- Packet
 - IPHeader, 61
- SFC_Router
 - SFC_Payload, 46
- TCP
 - tcp_opt, 36
 - TCP_Payload, 35
- TCP, 35, 62
 - ack_received(), 39
 - DEFAULT_AWND_SIZE, 35
 - get_new_packet(), 40
 - in(), 38
 - log(), 39
 - out(), 39
 - pac_rcv, 38
 - pac_snt, 37
 - packet_received(), 40
 - print_log(), 41
 - print_trace, 38
 - resend_packet(), 40
 - service(), 39
 - start(), 39
 - TCP(), 38
 - tcp_bound_rto(), 41
 - TCP_INIT_SS_THRESH, 35
 - TCP_INITIAL_RTO, 35
 - TCP_MAX_DUP_ACKS, 35
 - tcp_opt, 36
 - TCP_Payload, 35
 - TCP_RTO_MAX, 35
 - TCP_RTO_MIN, 35
 - tcp_rtt_estimator(), 40
 - TCP_SERVICE_INTERVAL, 35
 - tcp_set_rto(), 40
 - tp, 37
 - update_w_size(), 39
- time out, 39, 40, 54
- Time To Live (TTL), 43
- time-stamp, 39
- TP_general, 27, 29
 - cur_state, 28
 - lost(), 29
 - state_time, 28
 - states, 28
 - time(), 29
 - TP_general(), 29
 - trans_time, 29
 - transmit(), 29
- TP_two_state, 26
 - CHAN_STATE, 26
 - cur_state, 26
 - lost(), 27
 - p_param, 26
 - q_param, 26
 - rate_bad, 26
 - rate_good, 26
 - state_time, 26
 - time(), 27
 - TP_two_state(), 27
 - trans_time, 27
 - transmit(), 27
- Traffic Source, 14, 17, 21–25, 33, 34, 38, 40, 59
- Traffic_Source, 32
 - get(), 34
- Traffic_source, 21–24, 31, 38
 - buff, 21
 - get(), 21, 22, 40
 - in(), 22
 - out(), 22
 - queue_packet(), 22
 - service(), 22
 - Traffic_source(), 21
- transmission
 - rate, 17, 18, 26
 - state, 26
 - time, 21, 27–31
 - window, 31
- Transmission Control Protocol, *see* TCP
- transmission rate, 17
- transmission time, 25
- Transmit Process, 20, 26–28
- Transmit_process, 20, 25
 - time(), 25
 - transmit(), 21, 25
- TS_Exp, 21, 22
 - get(), 23
 - log(), 24
 - next_dest(), 24

- output_node, 23
- p_size, 23
- sep_mean, 23
- TS_Exp(), 23
- TS_Greedy, 21, 22, 24
 - get(), 25
 - log(), 25
 - next_dest(), 25
 - p_size, 24
 - TS_Greedy(), 24
- Vertex, 44, 45
 - cost(), 45
 - id(), 45
 - operator<(), 46
 - operator==(), 46
 - operator>(), 46
 - parent_branch(), 46
 - the_cost, 45
 - the_id, 45
 - the_parent_branch, 45
 - Vertex(), 45
- virtual, 14, 17, 31
- virtual functions
 - Generator
 - in(), 32
 - log(), 33
 - out(), 33
 - service(), 33
 - start(), 33
 - Link
 - busy(), 18
 - in(), 18
 - log(), 19
 - out(), 18
 - service(), 19
 - start(), 18
 - stop(), 18
 - Node
 - busy(), 15
 - initialise(), 16
 - next_dest(), 16
 - out_event(), 15
 - start(), 15
 - stop(), 16
 - Router
 - in(), 43
 - initialise(), 43
 - log(), 43
 - out(), 43
 - out_event(), 43
 - service(), 43
- window flow control, 31, 35
- window size, 31, 32, 35, 38
- world file, 59