

# Using the Georgia Tech Network Simulator<sup>1</sup>

Dr. George F. Riley

Department of Electrical and Computer Engineering  
Georgia Institute of Technology

The Georgia Tech Network Simulator (*GTNets*) is designed to allow network researchers to conduct simulation-based experiments to observe the behavior of moderate to large scale computer networks under a variety of conditions. The *GTNets* environment allows the creation of simulation network topologies (consisting of nodes and their associated communication links), and end-user applications describing the flow of data over the simulated topology.

This manual describes how to use the *GTNets*, both by using the features and network protocols implemented in the baseline *GTNets* package, and by modifying and extending the basic functionality to include new or enhanced protocols as defined by researchers.

---

<sup>1</sup>The Georgia Tech Network Simulator is an extensive collaboration between George Riley and his students at Georgia Tech. The research to develop and maintain this simulation tool is funded in part by the Defense Advanced Research Projects Agency (DARPA) under contract number N66002-00-1-8934 and by the National Science Foundation (NSF) under grant numbers ANI-9977544, ANI-0225417 and ECS-0225471. All opinions and conclusions expressed herein are exclusively those of the author, and do not necessarily represent the views of DARPA or NSF.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>5</b>
<b>3</b>	<b>The Simulator Object</b>	<b>10</b>
<b>4</b>	<b>Defining Topologies</b>	<b>18</b>
4.1	Nodes . . . . .	18
4.2	Interfaces . . . . .	39
4.3	Queues . . . . .	41
4.4	Links . . . . .	49
4.5	Stock Objects . . . . .	68
4.5.1	GTitm Modeller . . . . .	68
4.5.2	TransitNode . . . . .	69
4.5.3	SubDomain . . . . .	71
4.5.4	Star Topology . . . . .	71
4.5.5	Grid Topology . . . . .	74
4.5.6	Dumbbell Topology . . . . .	76
4.5.7	Tree Topology . . . . .	81
4.5.8	Random Topology . . . . .	83
<b>5</b>	<b>Defining Applications</b>	<b>84</b>
5.1	Application Base Class . . . . .	84
5.2	CBRApplcation . . . . .	87
5.3	OnOffApplication . . . . .	88
5.4	TCPApplication . . . . .	89
5.5	TCPReceive . . . . .	90
5.6	TCPServer . . . . .	91
5.7	WebServer . . . . .	92
5.8	WebBrowser . . . . .	93
<b>6</b>	<b>Layer 4 Protocols</b>	<b>96</b>
6.1	L4Protocol Base Class . . . . .	96
6.2	TCP Protocol Base Class and Variations . . . . .	101
6.3	Round-Trip Time Estimators . . . . .	108
6.4	UDP Protocol . . . . .	111
<b>7</b>	<b>Layer 3 Protocols</b>	<b>112</b>
<b>8</b>	<b>Layer 2 Protocols</b>	<b>114</b>
8.1	IEEE 802.2 . . . . .	114
8.2	IEEE 802.11 . . . . .	115

<b>9 Routing</b>	<b>116</b>
9.1 Routing Base Class . . . . .	116
9.2 Nix-Vector Routing . . . . .	118
9.3 Static Routing . . . . .	120
9.4 Manual Routing . . . . .	122
9.5 Wireless Dynamic Source Routing . . . . .	123
<b>10 Tracing Packets</b>	<b>124</b>
10.1 Trace File Object . . . . .	125
<b>11 Miscellaneous Support Objects</b>	<b>128</b>
11.1 Random Number Generators . . . . .	128
11.1.1 Random Number Base Class . . . . .	128
11.1.2 Uniform Random . . . . .	129
11.1.3 Constant Random . . . . .	130
11.1.4 Sequential Random . . . . .	130
11.1.5 Exponential Random . . . . .	131
11.1.6 Pareto Random . . . . .	131
11.1.7 Empirical Random . . . . .	132
11.1.8 Integer Empirical Random . . . . .	132
11.2 Statistics Gathering . . . . .	133
11.2.1 Statistics Base Class . . . . .	133
11.2.2 Histogram Class . . . . .	133
11.2.3 Average-Min-Max Class . . . . .	134
11.3 Packets and PDUs . . . . .	135
11.3.1 Packets . . . . .	135
11.3.2 Protocol Data Units . . . . .	138
11.4 Command Line Argument Processing . . . . .	140
11.5 IP Address Management Object . . . . .	143
11.6 Time and Rate Parsing Objects . . . . .	143
11.6.1 Time Parsing Object . . . . .	144
11.6.2 Rate Parsing Object . . . . .	144

# Chapter 1

## Introduction

The Georgia Tech Network Simulator (*GTNets*) is a full-featured network simulation environment that allows researchers in computer networks to study the behavior of moderate to large scale networks, under a variety of conditions. The design philosophy of *GTNets* is to create a simulation environment that is structured much like actual networks are structured. For example, in *GTNets*, there is clear and distinct separation of protocol stack layers. Packets in *GTNets* consist of a list of *protocol data units (PDUs)* that are appended and removed from the packet as it moves down and up the protocol stack. Simulation objects representing network nodes have one or more *Interfaces*, each of which can have an associated *IP Address* and an associated link. Layer 4 protocol objects in *GTNets* are bound to ports, in a fashion nearly identical to the binding to ports in real network protocols. Connections between protocol objects at the transport layer are specified using a source IP, source port, destination IP, destination port tuple just like actual *TCP* connections. The interface between applications and transport protocols uses the familiar *connect*, *listen*, *send*, and *sendto* calls much like the ubiquitous *sockets* API in Unix environments. Applications in *GTNets* can have one or more associated protocol objects, and can simulate the flow of data (including actual data contents) between applications.

A partial list of the features and capabilities of the *GTNets* simulator is given below.

- Connection Oriented Applications. Supports a large variety of TCP-based applications, in a client/server environment, including FTP models, and web-browsing models.
- Connectionless Applications. Models for UDP based connectionless applications including On-Off data sources and constant bit rate sources.
- TCP. Models Tahoe, Reno, NewReno, and SACK. Each *TCP* model supports detailed logging of sequence vs. time plots for both sequence and acknowledgement numbers.
- Routing. Routes can be either calculated statically, on-demand using the NixVector approach, or manually by the simulation user.
- Node Mobility. Supports node mobility using both random waypoint and specific waypoint models.
- Random Number Generators. Contains models for a variety of random number generators, including exponential, pareto, uniform, normal, empirical, constant, and sequential.
- Packet Tracing. Supports very fine-grained control over the tracing of packets through the simulation. Tracing can be enabled or disabled by node, protocols, or specific protocol endpoints. Furthermore, individual data items in each protocol header can be selectively enabled or disabled from being logged.
- Layer 3 Protocols. Supports *IP* version 4.
- Layer 2 Protocols. Supports both IEEE 802.3 and IEEE 802.11 protocols.
- Links. Supports Point-to-Point, Shared Ethernet, Switched Ethernet, and Wireless links.
- Queuing. Supports the drop-tail, Random Early Detection (*RED*), and Infinitesimal Perturbation Analysis (*IPA*) queuing methods.

- Statistics gathering. Supports data collection using histograms and cumulative distribution functions.
- Animation. Support graphical viewing of the simulation topology, with selective enabling and disabling of display for specified nodes and links.
- Stock Topology Objects. Supports a number of *stock objects* for topology generation, including Star, Tree, Dumbbell, and Grid.
- Distributed Simulations. Supports distributing a single simulation topology on either a network of loosely coupled workstations, a shared-memory symmetric multiprocessing system, or a combination of both.
- Simulation Statistics. Gathers and reports a large number of statistics regarding the performance of the simulator itself, including total number of events, total packets generated, total execution time, just to name a few.

## Chapter 2

# Overview

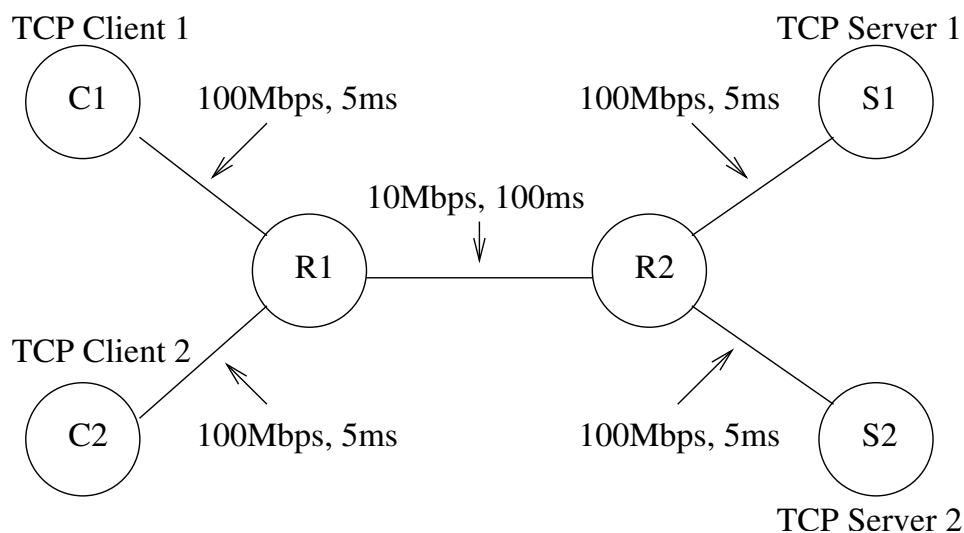


Figure 2.1: Sample GTNetS Topology

The *GTNets* simulator consists of a large number of *C++* objects which implement the behavior of a variety of network elements. Building and running a simulation using *GTNets*, requires creating a *C++* main program that instantiates the various network elements to describe a simulated topology, and the various applications and protocols used to move simulated data through the topology. The *C++* main program is then compiled with any compiler that fully complies with the *C++* standard<sup>1</sup>. After successfully compiling the main program, it is linked with the *GTNets* object libraries (on Linux these are available both *.a* and *.so* format and on Windows they are a *.lib* file), which creates an executable binary. The resulting binary is simply executed as any other application, which results in the simulation of the topology and data flows specified in the main program.

In this chapter, we give a small example of using *GTNets* to create a simple topology and two data flows. An example *GTNets* simulation is given for the simple topology shown in figure 2.1, and discussed in detail. In this simulation, there are six nodes, and two TCP flows from clients to servers. Many of the *GTNets* features will be used in this example, all of which are described in more detail later in this manual. The main program for the example is shown in program listing 2-1 starting on the following page. Each line of the sample program will be discussed briefly to describe its purpose in the simulation. After reading through this example, you should have a basic understanding of the various functions provided by *GTNets* and how to use them.

<sup>1</sup> *GTNets* has been compiled successfully on Linux with g++-2.96, g++-3.x, Microsoft Windows with MSVC-7.0, and Sun Solaris with SUNWS-CC version??

```
1 // Simple GNetS example
2 // George F. Riley, Georgia Tech, Winter 2002
3
4 #include "simulator.h" // Definitions for the Simulator Object
5 #include "node.h" // Definitions for the Node Object
6 #include "linkp2p.h" // Definitions for point-to-point link objects
7 #include "ratetimestruct.h" // Definitions for Rate and Time objects
8 #include "application-tcpserver.h" // Definitions for TCP Server application
9 #include "application-tcpsend.h" // Definitions for TCP Sending application
10 #include "tcp-tahoe.h" // Definitions for TCP Tahoe
11
12 int main()
13 {
14 // Create the simulator object
15 Simulator s;
16
17 // Create and enable IP packet tracing
18 Trace* tr = Trace::Instance(); // Get a pointer to global trace object
19 tr->IPDotted(true); // Trace IP addresses in dotted notation
20 tr->Open("intro1.txt"); // Create the trace file
21 TCP::LogFlagsText(true); // Log TCP flags in text mode
22 IPV4::Instance()->SetTrace(Trace::ENABLED); // Enable IP tracing all nodes
23
24 // Create the nodes
25 Node c1; // Client node 1
26 Node c2; // Client node 2
27 Node r1; // Router node 1
28 Node r2; // Router node 2
29 Node s1; // Server node 1
30 Node s2; // Server node 2
31
32 // Create a link object template, 100Mb bandwidth, 5ms delay
33 Linkp2p link(Rate("100Mb"), Time("5ms"));
34 // Add the links to client and server leaf nodes
35 c1.AddDuplexLink(&r1, link, IPAddr("192.168.0.1")); // c1 to r1
36 c2.AddDuplexLink(&r1, link, IPAddr("192.168.0.2")); // c2 to r1
37 s1.AddDuplexLink(&r2, link, IPAddr("192.168.1.1")); // s1 to r2
38 s2.AddDuplexLink(&r2, link, IPAddr("192.168.1.2")); // s2 to r2
39
40 // Create a link object template, 10Mb bandwidth, 100ms delay
41 Linkp2p r(Rate("10Mb"), Time("100ms"));
42 // Add the router to router link
43 r1.AddDuplexLink(&r2, r);
44
45 // Create the TCP Servers
46 TCP Server* server1 = (TCP Server*)s1.AddApplication(TCP Server(TCPTahoe()));
47 if (server1)
48 {
49 server1->BindAndListen(80); // Application on s1, port 80
50 server1->SetTrace(Trace::ENABLED); // Trace TCP actions at server1
51 }
52
53 TCP Server* server2 = (TCP Server*)s2.AddApplication(TCP Server(TCPTahoe()));
54 if (server2)
55 {
56 server2->BindAndListen(80); // Application on s2, port 80
```

Program 2-1 intro1.cc

```
57     server2->SetTrace(Trace::ENABLED); // Trace TCP actions at server2
58 }
59
60 // Set random starting times for the applications
61 Uniform startRv(0.0, 2.0);
62
63 // Create the TCP Sending Applications
64 TCPSend* client1 = (TCPSend*)c1.AddApplication(
65     TCPSend(s1.GetIPAddr(), 80,
66         Uniform(1000,10000),
67         TCPTahoe()));
68 if (client1)
69 {
70     // Enable TCP trace for this client
71     client1->SetTrace(Trace::ENABLED);
72     // Set random starting time
73     client1->Start(startRv.Value());
74 }
75
76 TCPSend* client2 = (TCPSend*)c2.AddApplication(
77     TCPSend(s2.GetIPAddr(), 80,
78         Constant(100000),
79         TCPTahoe()));
80 if (client2)
81 {
82     // Enable TCP trace for this client
83     client2->SetTrace(Trace::ENABLED);
84     // Set random starting time
85     client2->Start(startRv.Value());
86 }
87
88 s.Progress(1.0);           // Request progress messages
89 s.StopAt(10.0);          // Stop the simulation at time 10.0
90 s.Run();                  // Run the simulation
91 std::cout << "Simulation Complete" << std::endl;
92 }
```

Program 2-1 intro1.cc (continued)

**Include Files.** Lines 4 – 10 use the C/C++ `include` directive to include the definitions for the various network elements and simulation objects used in this simulation. The necessary include files will of course vary from simulation to simulation depending on which of the *GTNets* objects are used in the simulation. A complete listing of all *GTNets* objects and their corresponding include files is given in Appendix 11.6.2.

**Main Program.** The C++ main entry point is defined in line 12. All *GTNets* simulations must have a C++ main function.

**Simulator Object.** A single object of class `Simulator` must be created by all *GTNets* simulations before any other *GTNets* objects are created. In our example, the `Simulator` object is created at line 15. Chapter 3 describes all member functions available in the class `Simulator`.

**Defining the Trace File.** Lines 17 – 22 specify the name of the trace file and the desired level of tracing. For all *GTNets* simulations, there is a single global object of type `Trace` that manages all aspect of packet tracing in the simulation. Line 18 uses the `Trace::Instance()` function to obtain a pointer to the global trace object. Line 19 specifies that all *IP Addresses* should be written to the trace file in *dotted* notation, such as 192.168.0.1. The default notation for logging *IP Addresses* is 32 bit hexadecimal. Line 20 opens the actual trace file and assigns the name `intro1.txt`. Line 21 specifies that the flags field for all *TCP* headers logged in the trace file should use a text based

representation for the flags (such as SYN|ACK), rather than a 8 bit hexadecimal value. Line 22 specifies that all *IPV4* headers should be logged in the trace file for every packet received and every packet transmitted at all nodes. Complete details on specifying trace information can be found in chapter 10.

**Create Simulated Nodes.** Lines 24 – 30 create the node objects representing the six network nodes in the sample topology. In *GTNets*, node objects represent either end-systems (such as desktop systems or web servers), routers, or hubs. Notice that when creating `Node` objects in this example, we used statically defined `Node` objects, rather than dynamically allocating them with the *C++* `new` operator. Keep in mind that any `Node` object created as part of the topology *must* remain valid for the lifetime of the simulation. In this case, the nodes are created in the `main` function, which of course does not exit until the simulation is complete, and thus our nodes remain valid until then. However, if the nodes are created in a subroutine (such as `int CreateNodes() {}`), they would not persist after the `CreateNodes` function completed unless dynamically allocated with `new`.

**Create Simulated Links.** Lines 32 – 43 create the five link objects in the sample topology. First, line 33 creates a point-to-point link object of class `Linkp2p`. There are three things of interest in this declaration. First are two arguments to the constructor for `Linkp2p` objects, which specify the link bandwidth and propagation delay. The arguments are of type `Rate_t` and `Time_t` respectively, which are both of type `double`. However, in *GTNets*, anytime a variable of type `Rate_t` is required, an object of class `Rate` may be used instead. Objects of class `Rate` require a single argument in the constructor, which is a string value specifying rates using commonly recognized abbreviations for multipliers (such as Mb). Similarly, anytime a variable of type `Time_t` is required, an object of class `Time` may be used instead. Objects of class `Time` require a single argument in the constructor, which is a string value specifying rates using commonly recognized abbreviations for multipliers (such as ms). Complete details on the `Rate` and `Time` objects are given in chapter 11.

Secondly, notice that the object `link` is statically allocated in this case (it is not allocated using the `new` operator), and will be destroyed when the enclosing subroutine exits. This is the accepted way of defining and parameterizing links, and will be discussed in more detail below. Lines 35 – 38 specify the links connecting the clients to router `r1` and the servers to router `r2`. `Node` objects have a method `AddDuplexLink` which creates links between nodes. In this example, there are three arguments to `AddDuplexLink`, the `Node` pointer for the opposite link endpoint, the link object itself (`link` in this case), and an *IP Address* for the link endpoint. It is important to note that the `AddDuplexLink` method makes a *copy* of the `Link` object passed as the second parameter, rather than using it directly. Thus a single link object (`link` in this example) can be passed to any number of `AddDuplexLink` calls, and can subsequently be destroyed (when the subroutine exits) without causing problems in the simulation.

Finally notice that the third argument to `AddDuplexLink` in this example is the *IP Address* of the local link endpoint. The argument must be of type `IPAddr_t`, which is defined by *GTNets* to be `unsigned long`. However, in *GTNets*, whenever a variable of type `IPAddr_t` is needed, an object of class `IPAddr` may be used instead. Objects of class `IPAddr` require a single argument in the constructor, specifying the desired *IP Address* in the familiar dotted notation. Lines 40 – 43 create the slower speed 10Mb link object connecting the two routers, `r1` and `r2`. Complete details on creating and using `Link` objects are given in chapter 4.

**Create the TCP Servers.** The *TCP* server objects are created in lines 45 – 58. Lines 46 and 53 create two objects of class `TCPServer`, using the `AddApplication` method of a `Node` object. The `AddApplication` method accepts a single argument, which is a reference to any subclass of `Application`. The `Application` object that is passed to the `AddApplication` method is *copied*, and a pointer to the copied object is returned. For distributed simulations, the return value is possibly `NULL`, but this is discussed elsewhere in this manual. In this example, the subclass of `Application` that is used is class `TCPServer`, which creates an application that listens for connection requests on a specified port number. The constructor for `TCPServer` objects has a single parameter of class `TCP`, which specifies the variant of *TCP* to be used for this server object. Notice that, in this example, an anonymous temporary object `TCPTahoe()` is passed as the argument to the `TCPServer` constructor. The constructor for `TCPServer` makes a copy of the supplied object, rather than using it directly. The anonymous temporary `TCPTahoe` object is destroyed when the constructors at lines 46 and 53 are complete. Also, the argument to the `TCPServer` constructor can be omitted, in which case a copy of the default `TCP` object is used. Lines 49 and 56 assign the `TCPServer` objects to nodes `s1` and `s2` respectively, and bind to port 80. Lines 50 and 57 specify that all *TCP* headers should be logged in the trace file for all received and transmitted packets for these *TCP* endpoints.

**Create the TCP Clients.** The *TCP* client applications are created in lines 63 – 86. The objects of class `TCPSend` are created using the `AddApplication` method of `Node` objects, as previously discussed. The `TCPSend` application connects to a specified server and sends the specified amount of data. `TCPSend` constructors have four parameters, as follows. The first and second arguments are the *IP Address* and port number of the *TCP* server to which a connection is to be made. The second argument in this example illustrates the `GetIPAddr` method for `Node` objects, which returns the *IP Address* of the node (or the first *IP* of the node has more than one). The third parameter is a temporary object of class `Random` (or any subclass of `Random`), which specifies a random variable that determines how much data to send to the server. The first client specifies a `Uniform` random variable at line 65 which returns a uniform value between 1000 and 10,000. The second client specifies a `Constant` random variable at line 77, that returns the constant value 100,000. Details on the use of random variables in *GTNets* are given in chapter 11. The last parameter is a reference to a temporary object of class `TCP` (or any subclass of `TCP`) that specifies the *TCP* variation to be used for this *TCP* client. Notice that in this case, an anonymous temporary object of class `TCPTahoe` is specified. This parameter can be omitted, in which case a copy of the default `TCP` object is used.

Lines 70 and 82 enable tracing of the *TCP* headers for all packets sent and received by these endpoints.

**Start the Client Applications.** Lines 72 and 84 tell the simulator to create the connections between the clients and servers and start the sending applications. Line 60 defines a statically allocated `Uniform` random variable that will return random values uniformly in the interval [0.0, 2.0). Lines 72 and 84 use the `Start` method common to all *TCP* applications that specifies when the application should create the connection and begin sending the data.

**Start the Simulation.** Line 88 uses the `Progress` method of class `Simulator` to request a message be printed on `stdout` every 1.0 seconds of simulation time, indicating the simulation is progressing in time. Line 89 calls the `StopAt` method of `Simulator` to tell the simulation when to stop. Line 90 calls the `Run` method of `Simulator`, to run the simulation. The `Run` method does not exit until the simulation completes.

## Chapter 3

# The Simulator Object

As discussed in the overview chapter, all *GTNets* simulations must define exactly one instance of an object of class `Simulator`. This object is the basic simulation engine, and controls the scheduling and executing of simulation events. The methods and members of the `Simulator` object are presented below. Class `Simulator`. Class `Simulator` defines the simulator object required for all *GTNets* simulations. The simulator object controls all actions once the simulation has started executing. Each *GTNets* simulation should declare a single `Simulator` object, usually as a local variable in the `main` program.

### `Simulator` Constructors:

`Simulator()`

The constructor for the `Simulator` object takes no arguments unless using the distributed simulation feature of *GTNets*.

---

`Simulator(SystemId_t)`

When using distributed simulations, the `Simulator` constructor should have a single argument specifying the simulator id, in the range  $[0..nProcs)$ , where `nProcs` is the number of processors in the distributed simulation.

#### Arguments:

Type	Description
<code>SystemId_t</code>	Specifies the simulator identifier for this process in the distributed simulation.

---

### `Simulator` Public Methods:

`Handle(Event*, Time_t)`

The `Simulator` class is a subclass of `Handler`, since the simulator object must handle events, specifically the progress events and the halt event.

#### Arguments:

Type	Description
<code>Event*</code>	A pointer to the event being handled.
<code>Time_t</code>	The current simulation time.

---

`Cancel(Event*)`

Cancels a pending event.

#### Arguments:

Type	Description
<code>Event*</code>	A pointer to the event to be canceled. When scheduling events that will later be canceled, a pointer to the event must be retained to allow cancellation.

---

```
Schedule(Event*)
```

Schedules a new event. The simulation time for the event must be stored in the event class, member `time`, by the caller prior to calling `Schedule`. The handler for the event is `caller`, which must be a subclass of `Handler`

**Arguments:**

Type	Description
Event*	A pointer to the event being scheduled.

---

```
Schedule(Event*, Time_t)
```

Schedules a new event. The simulation time for the event is specified in the argument list and will be stored in the event class, member `time`, by this method. The handler for the event must be already specified in the passed `Event`.

**Arguments:**

Type	Description
Event*	A pointer to the event being scheduled.
Time_t	A amount of time <i>in the future</i> for this event. Note the time is a relative time from the current simulation time. The actual time for the event is computed by adding the current simulation time to the value specified.

---

```
Schedule(Event*, Time_t, Handler*)
```

Schedules a new event. The simulation time for the event is specified in the argument list and will be stored in the event class, member `time`, by this method. The handler for the event is also specified in the argument list.

**Arguments:**

Type	Description
Event*	A pointer to the event being scheduled.
Time_t	A amount of time <i>in the future</i> for this event. Note the time is a relative time from the current simulation time. The actual time for the event is computed by adding the current simulation time to the value specified.
Handler*	A pointer to any object that is a subclass of class <code>Handler</code> , specifying the event handler for this event.

---

```
Schedule(Event*, Time_t, Handler&)
```

Schedules a new event. The simulation time for the event is specified in the argument list and will be stored in the event class, member `time`, by this method. The handler for the event is also specified in the argument list.

**Arguments:**

Type	Description
Event*	A pointer to the event being scheduled.
Time_t	A amount of time <i>in the future</i> for this event. Note the time is a relative time from the current simulation time. The actual time for the event is computed by adding the current simulation time to the value specified.
Handler&	A reference to any object that is a subclass of class <code>Handler</code> , specifying the event handler for this event.

---

```
Progress(Time_t)
```

Request a progress message be printed on `stdout` periodically. The message is printed either by a *GTNets* supplied progress method, or by a method supplied using the `ProgressHook` method of class `Simulator`

**Arguments:**

Type	Description
Time_t	The progress interval, specified in seconds. Note that this specifies the progress method is called periodically in <i>Simulation Time</i> , not in wall-clock time.

---

SimulatorEvent\* NodeDownAt(Node\*, const Random&)  
 Schedule a node failure at a specified time.

**Return Value:**

Type	Description
SimulatorEvent*	Scheduled event (in case caller wants to cancel)

**Arguments:**

Type	Description
Node*	Node to fail.
const Random&	A random variable specifying time to fail.

---

SimulatorEvent\* NodeUpAt(Node\*, const Random&)  
 Schedule a node recovery at a specified time.

**Return Value:**

Type	Description
SimulatorEvent*	Scheduled event (in case caller wants to cancel)

**Arguments:**

Type	Description
Node*	Node to recover
const Random&	A random variable specifying time to recover

---

SimulatorEvent\* InterfaceDownAt(Interface\*, const Random&)  
 Schedule a node failure at a specified time.

**Return Value:**

Type	Description
SimulatorEvent*	Scheduled event (in case caller wants to cancel)

**Arguments:**

Type	Description
Interface*	Interface to fail.
const Random&	A random variable specifying time to fail.

---

SimulatorEvent\* InterfaceUpAt(Interface\*, const Random&)  
 Schedule a node recovery at a specified time.

**Return Value:**

Type	Description
SimulatorEvent*	Scheduled event (in case caller wants to cancel)

**Arguments:**

Type	Description
Interface*	Interface to recover
const Random&	A random variable specifying time to recover

---

Event\* DeQueue()

Removes the earliest pending event from the event queue and returns a pointer to it.

**Return Value:**

Type	Description
Event*	A pointer to the earliest pending event in the event queue.

---

Event\* PeekEvent()

Returns a pointer to the earliest pending event in the event queue, but does not remove it from the queue.

**Return Value:**

Type	Description
Event*	A pointer to the earliest pending event in the event queue.

---

```
PrintStats()
```

Prints some fairly detailed statistics about the simulation. Normally would be called only after the `Run` method has exited and the simulation has completed.

---

```
Run()
```

Initiates the simulation. The `Run` method should be called after all the topology and data flow information has been defined. `Run` will start processing events, and will continue until the `Stop` event is processed, the `Halt()` method is called, or until the event list becomes empty.

---

```
StopAt(Time_t)
```

Schedules a `Stop` event at the specified time. This will cause the `Run()` method to exit at the specified time, even if there are more pending unprocessed events.

**Arguments:**

Type	Description
<code>Time_t</code>	The simulation time to schedule the <code>Stop</code> event.

---

```
Halt()
```

Immediately stops processing events and causes the `Run()` method to exit.

---

```
Silent(bool)
```

Set silent mode on or off. When silent, the simulator produces no printed information on stdout.

**Arguments:**

Type	Description
<code>bool</code>	true if silent mode desired.

---

```
UseBackplane()
```

Inform the distributed simulation infrastructure to use the "Dynamic Simulation Backplane", which converts packet information to a generic format for exporting to a foreign simulator.

---

```
DeleteObject(Object*)
```

Schedules an event at the current simulation time which deletes the specified `Object`. Sometimes, a *GTNets* object will determine that it is no longer needed, but a *C++* object cannot delete itself. This allows an object to schedule its own deletion.

**Arguments:**

Type	Description
<code>Object*</code>	A pointer to the object to be deleted.

---

```
ProgressHook(ProgressHook_t)
```

Specifies a subroutine to call each time a `Progress` event is processed. The `Progress` events are scheduled periodically, at an interval specified by the `Progress` method. The `ProgressHook` method must be a static method, and must have a single `Time_t` argument and return `void`. When the `ProgressHook` method is called the argument passed is the current simulation time.

**Arguments:**

Type	Description
<code>ProgressHook_t</code>	The static method to call on each <code>Progress</code> interval.

---

```
AddNotify(NotifyHandler*, Time_t, void*)
```

Call a notification at the specified time in the future

**Arguments:**

Type	Description
<code>NotifyHandler*</code>	<code>NotifyHandler</code> object to call
<code>Time_t</code>	Time in future.
<code>void*</code>	Pointer to pass to the notifier.

---

CleanupOnExit(*bool*)

Specifies that the Simulator object should clean up all allocated memory objects when destructed. This would normally be used when debugging the simulator, to help locate memory leaks.

**Arguments:**

Type	Description
------	-------------

<i>bool</i>	A boolean specifying whether the Simulator object should cleanup when destructed.
-------------	---

**Default Value** is true

---

TopologyChanged()

Notify the simulator that a topology change has occurred.

---

QTWindow\* GetQTWindow()

Returns a pointer to the QT Window object

**Return Value:**

Type	Description
------	-------------

QTWindow*	Pointer to the QT Window object
-----------	---------------------------------

---

DisplayTopology()

Causes a graphical animation window to be opened with a visual display of the topology.

---

DisplayTopologyAndReturn()

Causes a graphical animation window to be opened with a visual display of the topology. Returns immediately.

---

UpdateTopology()

Causes the existing animation window to update the state of all animated objects.

---

StartAnimation(*Time\_t*, *bool*)

Starts an animation on at the specified time.

**Arguments:**

Type	Description
------	-------------

<i>Time_t</i>	The time to start the animation.
---------------	----------------------------------

<i>bool</i>	True if the animation should initially be paused. <b>Default Value</b> is true
-------------	--

---

StopAnimation(*Time\_t*)

Stops the animation and closes the animation window at the specified time.

**Arguments:**

Type	Description
------	-------------

<i>Time_t</i>	The time to stop the animation.
---------------	---------------------------------

---

PauseAnimation(*Time\_t*)

Pauses the animation at the specified time.

**Arguments:**

Type	Description
------	-------------

<i>Time_t</i>	The time to pause the animation.
---------------	----------------------------------

---

AnimationUpdateInterval(*Time\_t*)

Specifies the initial update interval for the animation display. The update interval is adjustable via a slider on the animation window.

**Arguments:**

Type	Description
------	-------------

<i>Time_t</i>	The simulation time between animation updates.
---------------	--

---

AnimateWirelessTx(bool)  
Specify detailed animation of wireless transmissions

**Arguments:**

Type	Description
bool	true if wireless animation desired

---

bool AnimateWirelessTx()  
Determine if wireless transmit animation selected

**Return Value:**

Type	Description
bool	true if wireless animation selected.

---

bool AddBackgroundMap(const std::string&, const RectRegion&)  
Add background map lines from the specified map file. The maps can be either in the CIA World Databank II format, or a simple file of lat/lon coordinates with a single header line specifying the number of points. The CIA database files are available on the GTNetS web page.

**Return Value:**

Type	Description
bool	True if successfully loaded

**Arguments:**

Type	Description
const std::string&	Name of map database file.
const RectRegion&	Bounding Region <b>Default Value</b> is RectRegion

---

bool PlaybackTraceFile(const char\*)  
GTNetS can animate a previously run simulation, from the trace file logged by that simulation. The topology MUST be identical to that of the original simulation.

**Return Value:**

Type	Description
bool	True if trace file successfully opened

**Arguments:**

Type	Description
const char*	Trace file name

---

bool StartPlayback()  
Starts a trace file playback animation

**Return Value:**

Type	Description
bool	True if successful.

---

CustomBackground(CustomBackground\_t)  
Specifies a custom background callback

**Arguments:**

Type	Description
CustomBackground_t	Pointer to custom background callback function

---

NodeSelectedCallback(NodeSelected\_t)  
Specifies a method to call when nodes are selected on the animation display.

**Arguments:**

Type	Description
NodeSelected_t	Callback function pointer.

---

EnableAnimationRecorder(*bool*)

Enables or Disables the recorder icon on the animation display. Recording takes huge amounts of disk space, so we only enable this when requested.

**Arguments:**

Type	Description
<i>bool</i>	True if recorder icon enabled.

---

RecorderMPEGSpeedup(*Mult\_t*)

When recording the MPEG file, the normal frame rate is 25fps. However, in a network simulation, this means 40ms between frames, and is much too long to make a meaningful simulation animation. Specifying the speedup increases the frames per second rate by the specified value. For example, a speedup of 10.0 (the default value) will result in 250 fps.

**Arguments:**

Type	Description
<i>Mult_t</i>	Desired animation recorder speedup.

---

*Time\_t* SetupTime()

Returns the amount of wall clock (CPU) time used to initialize the simulation. This includes all actions from the start of the `main` function and the call to `Simulator::Run()`.

**Return Value:**

Type	Description
<i>Time_t</i>	The CPU time used to initialize the simulation.

---

*Time\_t* RouteTime()

Returns the amount of wall clock (CPU) time used to calculate routing information for the simulation.

**Return Value:**

Type	Description
<i>Time_t</i>	The CPU time used to calculate routing information.

---

*Time\_t* RunTime()

Returns the amount of wall clock (CPU) time used by the event processing phase of the simulation.

**Return Value:**

Type	Description
<i>Time_t</i>	The CPU time used to by the event processing.

---

*Time\_t* TotalTime()

Returns the amount of wall clock (CPU) time used by the simulation.

**Return Value:**

Type	Description
<i>Time_t</i>	The CPU time used to by the entire simulation.

---

*Count\_t* TotalEventsProcessed()

Returns the count of total events processed by the simulation.

**Return Value:**

Type	Description
<i>Count_t</i>	The total number of events processed.

---

*Count\_t* TotalEventsScheduled()

Returns the count of total events scheduled by the simulation.

**Return Value:**

Type	Description
<i>Count_t</i>	The total number of events scheduled.

---

```
Count_t EventListSize()
```

Returns the current size of the event list, which is the number of scheduled, but unprocessed, events.

**Return Value:**

Type	Description
Count_t	The current size of the pending events list.

---

Simulator **Static Methods:**

```
Time_t Now()
```

Returns the current simulation time.

**Return Value:**

Type	Description
Time_t	The current simulation time.

---

```
SystemId_t SystemId()
```

Returns the system identifier for this process. This is only useful for distributed simulations where each simulator has a unique id.

---

```
Size_t ReportMemoryUsage()
```

Returns the total amount of memory (bytes) used by the simulation.

**Return Value:**

Type	Description
Size_t	The total number of memory bytes used by the simulation.

---

```
Size_t ReportMemoryUsageMB()
```

Returns the total amount of memory (MegaBytes) used by the simulation.

**Return Value:**

Type	Description
Size_t	The total number of memory MegaBytes used by the simulation.

---

Simulator **Public Members:**

Type	Name	Description
Count_t	totevs	Count of the total number of events that have been scheduled
Count_t	totrm	Count of the total number of events that have been removed
Count_t	totevp	Count of the total number of event processed
Count_t	totevc	Count of total number of event cancelled
Count_t	evlSize	Current size of pending event list

---

## Chapter 4

# Defining Topologies

Defining the *Topology* for the simulated network is usually the first step in creating a *GTNets* simulation. The topology consists of *Nodes* and *Links*. Nodes represent end-systems (desktop workstations, laptops, etc.) or routers. Links connect nodes together, and can be point-to-point links, local area networks (Ethernet), or Wireless. This chapter gives details of the *GTNets C++* objects representing these topology objects.

Additionally, *GTNets* defines a number of *Stock* topology objects, such as a *Star* and *Dumbbell* that encapsulate these commonly used topology objects in an easy to use object.

### 4.1 Nodes

The *GTNets C++ Node* object represents a node in the simulated topology. Nodes have a number of other objects associated with them, including:

1. A numeric *Node Identifier* assigned automatically by *GTNets* which is used internally for bookkeeping. This node ID is assigned sequentially starting from zero with the first node created, and increments by one for each node.
2. A list of one or more *Interface* objects, that represent the hardware interface to a communications link.
3. A list of one or more *Routing* objects that are used to route packets from this node.
4. An optional node *Location* in the X-Y plane.
5. An optional *Mobility* object, defining the mobility method used by this node.
6. An optional *Protocol Graph*, used to locate protocol objects by protocol layer number and protocol number.
7. A port demultiplexer, used to keep track of layer 2, 3, and 4 port and protocol usage.

The remainder of this section describes the `Node` object and some of the helper objects in detail.

**Class `Node`.** Objects of class `Node` are used in *GTNets* to represent all nodes in the simulated topology. Nodes represent end-systems, wireless devices, hubs, and routers. When a node is created, it automatically has an associated routing object (the default is NIX-Vector routing) and a layer 3 protocol object (the default is *IPV4*). After nodes are created, interfaces and links can be added to describe the connectivity of nodes, using the public members described below.

Node **Constructors:**

Node(SystemId\_t)

Creates a node in the simulated topology. The optional parameter is useful only for distributed simulations, and specifies which simulator is responsible for maintaining the state for this node. If the id specified matches the system id specified on the Simulator constructor, then a full-state node is created. If not, a reduced state node, with only connectivity information, is created.

**Arguments:**

Type	Description	Default Value
SystemId_t	The system responsible for this node (distributed simulations only).	is 0

---

Node **Public Methods:**

NodeId\_t Id()

Each node in a *GTNets* simulation has an associated unique integer identifier, starting from zero for the first node created.

**Return Value:**

Type	Description
NodeId_t	The node identifier for this node.

---

IPAddr\_t GetIPAddr()

Returns the *IP Address* assigned to this node. Nodes can have multiple *IP Address*s, since *IP Address*s are assigned to interfaces and nodes can have multiple interfaces. This returns the first *IP Address* assigned to this node.

**Return Value:**

Type	Description
IPAddr_t	The first <i>IP Address</i> assigned to this node.

---

SetIPAddr(IPAddr\_t)

Specifies an *IP Address* for this node.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> to be assigned to this node.

---

IPAddrs(IPMaskVec\_t&)

Each interface assigned to a node has an associated *IP Address* and address mask. This method returns a vector of the *IP Address* and masks for each interface on this node.

**Arguments:**

Type	Description
IPMaskVec_t&	An <i>out</i> parameter that is populated with the <i>IP Address</i> and mask information.

---

bool LocalIP(IPAddr\_t)

Determines if the specified *IP Address* is local to this node.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> to check for locality.

---

bool IPKnown()

Determines if this node has an *IP Address* assigned.

---

```
bool IsReal()
```

Find out if this node is "real", versus a Ghost

**Return Value:**

Type	Description
bool	True if Real node, false if Ghost.

---

```
bool IsSwitchNode()
```

Find out if this node is switch.

---

```
IsSwitchNode(bool)
```

Set the node to a switch or not

**Arguments:**

Type	Description
bool	

---

```
PacketRX(Packet*, Interface*)
```

called when a packet is receive on a switch node

**Arguments:**

Type	Description
Packet*	
Interface*	

---

```
Interface* AddInterface(const L2Proto&, bool)
```

Assign a new interface to this node, with no associated *IP Address* or mask.

**Return Value:**

Type	Description
Interface*	A pointer to the newly assigned interface.

**Arguments:**

Type	Description
const L2Proto&	A layer 2 protocol object to be assigned to this interface. The object passed in is <i>copied</i> , so the same object may be passed to multiple interfaces without problems. <b>Default Value</b> is L2Proto802_3
bool	<b>Default Value</b> is false

---

```
Interface* AddInterface(const L2Proto&, IPAddr_t, Mask_t, MACAddr, bool)
```

Assign a new interface to this node, with the specified *IP Address* and mask.

**Return Value:**

Type	Description
Interface*	A pointer to the newly assigned interface.

**Arguments:**

Type	Description
const L2Proto&	A layer 2 protocol object to be assigned to this interface. The object passed in is <i>copied</i> , so the same object may be passed to multiple interfaces without problems.
IPAddr_t	The <i>IP Address</i> assigned to this interface.
Mask_t	The address mask assigned to this interface.
MACAddr	The <i>MAC</i> address assigned (normally the default should be used) <b>Default Value</b> is MACAddr::Allocate
bool	<b>Default Value</b> is false

---

```
Interface* AddInterface(const L2Proto&, const Interface&, IPAddr_t, Mask_t,
MACAddr, bool)
```

Assign a new interface to this node, with the specified *IP Address* and mask.

**Return Value:**

Type	Description
Interface*	A pointer to the newly assigned interface.

**Arguments:**

Type	Description
const L2Proto&	A layer 2 protocol object to be assigned to this interface. The object passed in is <i>copied</i> , so the same object may be passed to multiple interfaces without problems.
const Interface&	Reference to what type of Interface to add
IPAddr_t	The <i>IP Address</i> assigned to this interface.
Mask_t	The address mask assigned to this interface.
MACAddr	The <i>MAC</i> address assigned (normally the default should be used) <b>Default Value</b> is MACAddr::Allocate
bool	<b>Default Value</b> is false

---

```
Count_t InterfaceCount()
```

Returns the number of interfaces assigned to this node.

**Return Value:**

Type	Description
Count_t	The number of interfaces assigned to this node.

---

```
const IFVec_t& Interfaces()
```

Returns a reference to a vector of all interfaces assigned to this node.

**Return Value:**

Type	Description
const IFVec_t&	A vector of interface pointers for each interfae assigned to this node.

---

```
Interface* GetIfByLink(Link*)
```

Returns the interface associated with the specified link.

**Return Value:**

Type	Description
Interface*	The interface pointer associated with the specified link. Returns <i>nil</i> if not found.

**Arguments:**

Type	Description
Link*	A pointer to the link object to search for.

---

```
Interface* GetIfByNode(Node*)
```

Returns the interface used to communicate with the specified node.

**Return Value:**

Type	Description
Interface*	A pointer to the interface used to communicate with the specified node. Returns <i>nil</i> if not found.

**Arguments:**

Type	Description
Node*	A pointer to the node to search for.

---

```
Interface* GetIfByIP(IPAddr_t)
```

Returns the interface with the specified *IP Address*.

**Return Value:**

Type	Description
Interface*	A pointer to the interface with the specified <i>IP Address</i> . Returns <i>nil</i> if not found.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> to find.

---

```
Broadcast(Packet*, Proto_t)
```

Broadcasts a packet on all associated interfaces.

**Arguments:**

Type	Description
Packet*	The packet to send. Each interface gets a separate copy of the packet, so the caller is still responsible for the memory for the packet, and should delete this packet when no longer needed.
Proto_t	The layer 3 protocol number to insert in the layer 2 header for this packet.

---

```
Interface* AddDuplexLink(Node*)
```

Adds a new duplex link (and an associated interface) to this node, using the default link type.

**Return Value:**

Type	Description
Interface*	A pointer to the newly added interface on the local node.

**Arguments:**

Type	Description
Node*	A node pointer to the peer node for this link.

---

```
Interface* AddDuplexLink(Node*, const Linkp2p&)
```

Adds a new duplex link (and an associated interface) to this node, using the specified link type.

**Return Value:**

Type	Description
Interface*	A pointer to the newly added interface on the local node.

**Arguments:**

Type	Description
Node*	A node pointer to the peer node for this link.
const Linkp2p&	The type of link to add. The passed argument is copied, so an anonymous temporary object can be passed, or the same link object can be passed multiple times.

---

```
Interface* AddDuplexLink(Interface*, Interface*)
```

Add a new duplex link between two existing interfaces, using the default link type.

**Return Value:**

Type	Description
Interface*	A pointer to the first interface specified.

**Arguments:**

Type	Description
Interface*	First existing interface.
Interface*	Second existing interface.

---

```
Interface* AddDuplexLink(Interface*, Interface*, const Linkp2p&)
```

Add a new duplex link between two existing interfaces, using the specified link object type.

**Return Value:**

Type	Description
Interface*	A pointer to the first interface specified.

**Arguments:**

Type	Description
Interface*	First existing interface.
Interface*	Second existing interface.
const Linkp2p&	The link object to use. This object is copied, so an anonymous temporary object can be used, or the same object can be passed multiple times.

---

```
Interface* AddDuplexLink(Node*, IPAddr_t, Mask_t, IPAddr_t, Mask_t)
```

Adds a new duplex link (and an associated interface) to this node, using the default link type, and specifying the *IP Address* and mask for both new interfaces.

**Return Value:**

Type	Description
Interface*	A pointer to the newly added interface on the local node.

**Arguments:**

Type	Description
Node*	A node pointer to the peer node for this link.
IPAddr_t	The <i>IP Address</i> for the local interface.
Mask_t	The address mask for the local interface. <b>Default Value</b> is Mask32
IPAddr_t	The <i>IP Address</i> for the remote interface. <b>Default Value</b> is IPADDR_NONE
Mask_t	The address mask for the remote interface. <b>Default Value</b> is Mask32

---

```
Interface* AddDuplexLink(Node*, const Linkp2p&, IPAddr_t, Mask_t, IPAddr_t, Mask_t)
```

Adds a new duplex link (and an associated interface) to this node, using the specified link type, and specifying the *IP Address* and mask for both new interfaces.

**Return Value:**

Type	Description
Interface*	A pointer to the newly added interface on the local node.

**Arguments:**

Type	Description
Node*	A node pointer to the peer node for this link.
const Linkp2p&	The type of link to add. The passed argument is copied, so an anonymous temporary object can be passed, or the same link object can be passed multiple times.
IPAddr_t	The <i>IP Address</i> for the local interface.
Mask_t	The address mask for the local interface. <b>Default Value</b> is Mask32
IPAddr_t	The <i>IP Address</i> for the remote interface. <b>Default Value</b> is IPADDR_NONE
Mask_t	The address mask for the remote interface. <b>Default Value</b> is Mask32

---

```
Interface* AddRemoteLink(IPAddr_t, Mask_t)
```

Used for distributed simulations only. Defines a link from this node to another node defined and managed on another simulator process. The default link bandwidth and link delay are used.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> for the local interface.
Mask_t	The address mask for the local interface.

---

```
Interface* AddRemoteLink(IPAddr_t, Mask_t, Rate_t, Time_t)
```

Used for distributed simulations only. Defines a link from this node to another node defined and managed on another simulator process.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> for the local interface.
Mask_t	The address mask for the local interface.
Rate_t	The bandwidth for the remote link.
Time_t	The propagation delay for the remote link.

---

```
Link* GetLink(Node*)
```

Returns a pointer to the link used to communicate with the specified node.

**Return Value:**

Type	Description
Link*	The link pointer for the link used to communicate with the specified node. Returns <i>nil</i> if none is found.

**Arguments:**

Type	Description
Node*	Node pointer to find a link for.

---

```
Interface* AddSimplexLink(Node*)
```

Add a simplex link from this node to the specified node, using the default link object and with no *IP Address* or mask.

**Return Value:**

Type	Description
Interface*	An interface pointer for the local interface added for this link.

**Arguments:**

Type	Description
Node*	The node pointer for the remote node.

---

```
Interface* AddSimplexLink(Node*, const Linkp2p&)
```

Add a simplex link from this node to the specified node, using the specified link object and with no *IP Address* or mask.

**Return Value:**

Type	Description
Interface*	An interface pointer for the local interface added for this link.

**Arguments:**

Type	Description
Node*	The node pointer for the remote node.
const Linkp2p&	The link object to copy for this simplex link.

---

```
Interface* AddSimplexLink(Node*, const Linkp2p&, IPAddr_t, Mask_t)
```

Add a simplex link from this node to the specified node, using the specified link object and with the specified *IP Address* and mask.

**Return Value:**

Type	Description
Interface*	An interface pointer for the local interface added for this link.

**Arguments:**

Type	Description
Node*	The node pointer for the remote node.
const Linkp2p&	The link object to copy for this simplex link.
IPAddr_t	The <i>IP Address</i> for the local interface.
Mask_t	The address mask for the local interface.

---

```
Queue* GetQueue()
```

Get a pointer to the queue object for this node. Since nodes can have multiple interfaces and queues, this is useful only for nodes with a single interface.

**Return Value:**

Type	Description
Queue*	A pointer to the single queue for this node. Returns <i>nil</i> if no interfaces, or more than one interface.

---

```
Queue* GetQueue(Node*)
```

Get a pointer to the queue object used to buffer packets destined for the specified node.

**Return Value:**

Type	Description
Queue*	A pointer to the queue object used to buffer packets destined for the specified node. Returns <i>nil</i> if none is found.

**Arguments:**

Type	Description
Node*	The target node to locate the queue for.

---

```
Application* AddApplication(const Application&)
```

Adds an application associated with this node.

**Return Value:**

Type	Description
Application*	A pointer to the newly added application.

**Arguments:**

Type	Description
const Application&	The application object to use. This application object is copied, so an anonymous temporary object can be specified, or the same application object can be used multiple times.

---

```
Neighbors(NodeWeightVec_t, bool)
```

Get a list of all neighbors for this node. An output parameter, returning a vector of nodes and link

**Arguments:**

Type	Description
NodeWeightVec_t	weights to each directly connected neighbor. True if skip leaf neighbors desired
bool	<b>Default Value</b> is false

---

```
Count_t NeighborCount()
```

Get a count of directly connected neighbors. //Doc:Arg1 The count of directly connected neighbors.

---

```
NeighborsByIf(Interface*, IPAddrVec_t&)
```

Get a list of *IP Address* neighbors connected using the specified interface.

**Arguments:**

Type	Description
Interface*	The interface to return the neighbors for.
IPAddrVec_t&	An output parameter giving a vector of <i>IP Address</i> s connected on the specified interface.

---

```
DefaultRoute(RoutingEntry)
```

Specifies the default route for packets leaving this node.

**Arguments:**

Type	Description
RoutingEntry	The routing entry (interface and <i>IP Address</i> ) to use when no other routing information can be found.

---

DefaultRoute(Node\*)

Specifies the default route for packets leaving this node.

**Arguments:**

Type	Description
Node*	The next hop neighbor to use for routing when no other routing information can be found.

---

AddRoute(IPAddr\_t, Count\_t, Interface\*, IPAddr\_t)

Add a new routing table entry.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> specifying the destination address this routing entry is for.
Count_t	The mask for the destination <i>IP Address</i> , which allows route aggregation.
Interface*	The interface for the next hop target for the specified destination.
IPAddr_t	The <i>IP Address</i> of the next hop target.

---

RoutingEntry LookupRoute(IPAddr\_t)

Look up the routing table entry for a specified destination *IP Address*.

**Return Value:**

Type	Description
RoutingEntry	The routing table entry for the specified destination. The interface pointer in this entry will be <i>nil</i> if no routing information is found for the specified destination.

**Arguments:**

Type	Description
IPAddr_t	The destination <i>IP Address</i> to look up.

---

RoutingEntry LookupRouteNix(Count\_t)

Look up the routing table entry by a neighbor index, used by the NIX-Vector routing method.

**Return Value:**

Type	Description
RoutingEntry	The routing table entry for the specified neighbor index. The interface pointer in this entry will be <i>nil</i> if no routing information is found for the specified destination.

**Arguments:**

Type	Description
Count_t	The neighbor index to look up.

---

Routing::RType\_t RoutingType()

Return the routing type in use at this node.

**Return Value:**

Type	Description
Routing::RType_t	The routing type in use. The enumerated type <i>RType_t</i> is defined in <i>routing.h</i> .

---

Interface\* LocalRoute(IPAddr\_t)

Determine if the specified *IP Address* is on a directly connected neighbor.

**Return Value:**

Type	Description
Interface*	An pointer to the interface used to communicate with the specified <i>IP Address</i> . Returns <i>nil</i> if the specified <i>IP Address</i> is not a directly connected neighbor.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> to search for.

---

```
InitializeRoutes()
```

Perform any initializations needed by the routing protocol used by this node.

---

```
Count_t RoutingFibSize()
```

Determine the size of the forwarding information base (FIB) at this node.

**Return Value:**

Type	Description
Count_t	The size, in bytes, of the FIB at this node.

---

```
Count_t GetNix(Node*)
```

Get the neighbor index for the specified node. Used by the NIX-Vector routing method.

**Return Value:**

Type	Description
Count_t	The neighbor index for the specified node. Returns MAX_COUNT if the specified node is not a neighbor.

**Arguments:**

Type	Description
Node*	The node to find the neighbor index for.

---

```
RoutingNixVector* GetNixRouting()
```

Get a pointer to the NIX-Vector routing object for this node.

**Return Value:**

Type	Description
RoutingNixVector*	A pointer to the NIX-Vector routing object for this node. Returns <i>nil</i> if NIX-Vector routing is not used at this node.

---

```
Routing* GetRouting()
```

Get a pointer to the routing object used by this node.

**Return Value:**

Type	Description
Routing*	Routing object pointer.

---

```
Protocol* LookupProto(Layer_t, Proto_t)
```

Find the protocol object at the specified protocol layer, and protocol number.

**Return Value:**

Type	Description
Protocol*	A pointer to the protocol object at the specified layer and protocol number.

**Arguments:**

Type	Description
Layer_t	The protocol layer number to look up.
Proto_t	The protocol number to look up.

---

```
InsertProto(Layer_t, Proto_t, Protocol*)
```

Inserts a new protocol in the protocol graph.

**Arguments:**

Type	Description
Layer_t	The protocol layer number to insert the protocol into.
Proto_t	The protocol number to insert the protocol into.
Protocol*	A protocol to insert.

---

```
bool Bind(Proto_t, PortId_t, Protocol*)
```

Bind a protocol to a port number at a particular protocol number.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
Proto_t	The protocol number for this port.
PortId_t	The port to bind to.
Protocol*	The protocol to bind to the port.

---

```
bool Bind(Proto_t, PortId_t, IPAddr_t, PortId_t, IPAddr_t, Protocol*)
```

Binds a protocol to a local *IP Address*, port, remote *IP Address*, port, at a particular protocol number.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
Proto_t	The protocol number for this binding.
PortId_t	The local port number.
IPAddr_t	The local <i>IP Address</i> .
PortId_t	The remote port number.
IPAddr_t	The remote <i>IP Address</i> .
Protocol*	The protocol to bind.

---

```
PortId_t Bind(Proto_t, Protocol*)
```

Bind a protocol to any available port.

**Return Value:**

Type	Description
PortId_t	The port number bound to.

**Arguments:**

Type	Description
Proto_t	The protocol number to use.
Protocol*	The protocol to bind.

---

```
bool Unbind(Proto_t, PortId_t, Protocol*)
```

Remove a port binding.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
Proto_t	The protocol number to use.
PortId_t	The currently bound port number.
Protocol*	The protocol to unbind.

---

bool Unbind(Proto\_t, PortId\_t, IPAddr\_t, PortId\_t, IPAddr\_t, Protocol\*)  
 Remove a port binding by local and remote port and *IP Address*.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
Proto_t	The protocol number to use.
PortId_t	Local port number.
IPAddr_t	Local <i>IP Address</i> .
PortId_t	Remote port number.
IPAddr_t	Remote <i>IP Address</i> .
Protocol*	Protocol to unbind.

---

Protocol\* LookupByPort(Proto\_t, PortId\_t)  
 Locate the protocol bound to a specified port and protocol number.

**Return Value:**

Type	Description
Protocol*	A pointer to the protocol bound to the specified port. Returns <i>nil</i> if no protocol is found at the specified protocol and port.

**Arguments:**

Type	Description
Proto_t	The protocol number.
PortId_t	The port to look up.

---

Protocol\* LookupByPort(Proto\_t, PortId\_t, IPAddr\_t, PortId\_t, IPAddr\_t)  
 Locate the protocol bound to the specified local and remote port and *IP Address*, at the specified protocol number.

**Return Value:**

Type	Description
Protocol*	A pointer to the protocol bound as specified. Returns <i>nil</i> if no protocol is found.

**Arguments:**

Type	Description
Proto_t	The protocol number.
PortId_t	Local port number.
IPAddr_t	Local <i>IP Address</i> .
PortId_t	Remote port number.
IPAddr_t	Remote <i>IP Address</i> .

---

bool TracePDU(Protocol\*, PDU\*, Packet\*, char\*)  
 Log the contents of the specified protocol data unit, from the specified protocol, to the trace file.

**Return Value:**

Type	Description
bool	Returns true if trace file exists.

**Arguments:**

Type	Description
Protocol*	The protocol pointer for the protocol object requesting the trace.
PDU*	The protocol data unit pointer.
Packet*	Packet that contains this PDU (optional) <b>Default Value</b> is nil
char*	Extra text information for trace file (optional) <b>Default Value</b> is nil

---

```
SetTrace(Trace::TraceStatus)
```

Specifies the tracing status of this node. The trace status desired. This can be either `Trace::ENABLED`, which indicates that all packets requesting tracing at this node will be traced; `Trace::DISABLED` indicating that no packets will be traced, or `Trace::DEFAULT`, indicating that the decision is deferred to the protocols at this node.

**Arguments:**

Type	Description
<code>Trace::TraceStatus</code>	The trace status desired for this node.

---

```
SetLocation(Meters_t, Meters_t, Meters_t)
```

Set the location of this node in the X–Y–Z coordinate.

**Arguments:**

Type	Description
<code>Meters_t</code>	The X–coordinate (meters).
<code>Meters_t</code>	The Y–coordinate (meters).
<code>Meters_t</code>	The Z–coordinate (meters). <b>Default Value</b> is 0

---

```
SetLocation(const Location&)
```

Set the location of this node in the X–Y–Z coordinate.

**Arguments:**

Type	Description
<code>const Location&amp;</code>	A <code>Location</code> object specifying the X–Y–Z coordinate (meters).

---

```
SetLocationLongLat(const Location&)
```

Set the location of this node in the X–Y plane.

**Arguments:**

Type	Description
<code>const Location&amp;</code>	

---

```
bool HasLocation()
```

Test if the node has a location assigned.

**Return Value:**

Type	Description
<code>bool</code>	True if location assigned for this node.

---

```
Meters_t LocationX()
```

Return the X–coordinate of this node.

**Return Value:**

Type	Description
<code>Meters_t</code>	The X–coordinate (meters) of this node.

---

```
Meters_t LocationY()
```

Return the Y–coordinate of this node.

**Return Value:**

Type	Description
<code>Meters_t</code>	The Y–coordinate (meters) of this node.

---

```
Meters_t LocationZ()
```

Return the Z–coordinate of this node.

**Return Value:**

Type	Description
<code>Meters_t</code>	The Z–coordinate (meters) of this node.

---

```
Location GetLocation()
```

Return the location of this node.

**Return Value:**

Type	Description
Location	A Location object specifying the X, Y and Z coordinates of this node (meters).

---

```
Location UpdateLocation()
```

The location of mobile nodes is not updated continually during the simulation. Rather, it is updated when UpdateLocation() is called.

**Return Value:**

Type	Description
Location	A Location object specifying the X, Y and Z coordinates of this node (meters).

---

```
Mobility* AddMobility(const Mobility&)
```

Add a mobility model to this node.

**Return Value:**

Type	Description
Mobility*	A pointer to the added mobility object.

**Arguments:**

Type	Description
const Mobility&	The Mobility object to use. This is copied to the node, so an anonymous temporary object can be used, or the same Mobility object can be passed multiple times.

---

```
Mobility* GetMobility()
```

Return existing mobility model.

**Return Value:**

Type	Description
Mobility*	Pointer to this nodes mobility model. NIL if none.

---

```
bool IsMobile()
```

Determine if this node has a mobility model.

**Return Value:**

Type	Description
bool	True if node has a mobility model.

---

```
bool IsMoving()
```

Determine if this node is presently moving (rather than paused or temporarily stationary).

**Return Value:**

Type	Description
bool	True if node is presently moving.

---

```
Meters_t Distance(Node*)
```

Calculate distance between this node and specified peer node.

**Return Value:**

Type	Description
Meters_t	Distance (meters).

**Arguments:**

Type	Description
Node*	Peer node pointer.

---

---

```
BuildRadioInterfaceList(void)
```

Build a list of nodes in transmission range.

**Arguments:**

Type	Description
void	

---

```
const RadioVec_t& GetRadioInterfaceList()
```

Get a reference to the current radio interface list

**Return Value:**

Type	Description
const RadioVec_t&	Reference to the current radio interface list

---

```
SetRadioRange(Meters_t)
```

Specify the power range of this wireless transmitter (meters).

**Arguments:**

Type	Description
Meters_t	Power range (meters)

---

```
Meters_t GetRadioRange(void)
```

Return the power range of this transmitter.

**Return Value:**

Type	Description
Meters_t	Power range (meters).

**Arguments:**

Type	Description
void	

---

AddCallback(Layer\_t, Proto\_t, PacketCallbacks::Type\_t, Interface\*, PacketCallbacks) Add a callback function to this node. Callbacks allow a function to examine every packet received by this node at any protocol layer, and either packet receipt or packet transmission

**Arguments:**

Type	Description
Layer_t	Layer to insert callback. Specifying zero indicates all layers will call the callback.
Proto_t	Protocol number for callback. Specifying zero indicates the callback should be called for all protocols at the specified layer.
PacketCallbacks::Type_t	Type of callback. Either PacketCallbacks::RX, indicating a callback on packet receipt, or PacketCallbacks::Tx indicating callback on packet transmission.
Interface*	Interface pointer. Layer 2 callbacks can specify an additional restriction specifying a particular interface only.
PacketCallbacks::Function_t	Callback function to add. The callback function must return true or false indicating that the packet should (true) or should not (false) continue processing by the caller. In other words, if the callback function drops the packet, it should return false. The callback function must accept five arguments: Layer_t, PacketCallbacks::Type_t, Packet* Node* and Interface*. The layer is the layer number from which the callback was generated, the Type_t is either RX or TX as described above, the Packet* is the packet, and the Node* is the node processing the packet. The Interface* is the interface processing the packet (layer 2 only)

---

DeleteCallback(Layer\_t, Proto\_t, PacketCallbacks::Type\_t, Interface\*) Remove a callback function from this node.

**Arguments:**

Type	Description
Layer_t	Layer to remove callback from.
Proto_t	Protocol number to delete.
PacketCallbacks::Type_t	Type or callback to remove, either TX or RX as above.
Interface*	Interface to remove callback from.

---

bool CallCallbacks(Layer\_t, Proto\_t, PacketCallbacks::Type\_t, Packet\*, Interface\*) Call any callback defined on this node for the layer and type specified.

**Arguments:**

Type	Description
Layer_t	Layer for callback.
Proto_t	Protocol number for callback.
PacketCallbacks::Type_t	Type for callback.
Packet*	Packet being processed.
Interface*	<b>Default Value</b> is nil

---

UserInformation(void\*)

Nodes have a blind pointer that can contain arbitrary information useful the the user's simulation.

**Arguments:**

Type	Description
void*	Pointer to any information. Will be stored in the node object, and returned as needed.

---

```
void* UserInformation()
```

Return the previously specified user information pointer.

**Return Value:**

Type	Description
void*	User information pointer (nil if none specified)

---

```
bool WirelessTx()
```

Check if the node is transmitting on a wireless link. Used by the animation widow.

**Return Value:**

Type	Description
bool	True if transmitting.

---

```
bool WirelessRx()
```

Check if the node is receiving on a wireless link. Used by the animation widow.

**Return Value:**

Type	Description
bool	True if receiving.

---

```
bool WirelessCx()
```

Check if the node is experiencing collision on a wireless link. Used by the animation widow.

**Return Value:**

Type	Description
bool	True if colliding.

---

```
bool WirelessRxMe()
```

Check if the node is receiving on a wireless link. Used by the animation widow.

**Return Value:**

Type	Description
bool	True if receiving.

---

```
bool WirelessRxZz()
```

Check if the node is receiving on a wireless link. Used by the animation widow.

**Return Value:**

Type	Description
bool	True if receiving.

---

```
Show(bool)
```

Enable or disable the animation window.

**Arguments:**

Type	Description
bool	True if animation window should be enabled, false if not.

---

```
bool Show()
```

Determine if the node should be displayed in the anim window

**Return Value:**

Type	Description
bool	True if the node should be displayed in the animation window

---

`QCanvasItem* Display(QTWindow*)`

Each node is responsible for "drawing" itself on the animation display. This allows the node to decide shape, size, etc.

**Return Value:**

Type	Description
<code>QCanvasItem*</code>	Any subclass of <code>QCanvasItem</code> .

**Arguments:**

Type	Description
<code>QTWindow*</code>	Qt Canvas to use. The node is drawn at present <code>Location()</code> .

---

`QCanvasItem* Display(const QPoint&, QTWindow*)`

Each node is responsible for "drawing" itself on the animation display. This allows the node to decide shape, size, etc.

**Return Value:**

Type	Description
<code>QCanvasItem*</code>	Any subclass of <code>QCanvasItem</code> .

**Arguments:**

Type	Description
<code>const QPoint&amp;</code>	Point in Qt canvas to display this node.
<code>QTWindow*</code>	Qt Canvas to use.

---

`WirelessTxColor(const QColor&)`

Specify color for the wireless transmit animation.

**Arguments:**

Type	Description
<code>const QColor&amp;</code>	Desired color.

---

`const QColor& WirelessTxColor()`

Get the specified wireless transmit color

**Return Value:**

Type	Description
<code>const QColor&amp;</code>	Color for wireless tx concentric circles

---

`bool PushWirelessTx(QCanvasItem*)`

Add a canvas item to the wireless transmit animation list

**Arguments:**

Type	Description
<code>QCanvasItem*</code>	Item (one of the concentric circles) to add

---

`QCanvasItem* PopWirelessTx()`

Pop and return the oldest item in the wireless tx history.

**Return Value:**

Type	Description
<code>QCanvasItem*</code>	Oldest item (nil if none).

---

`PixelSize(Count_t)`

Set the size of this node (pixels) on the animation display.

**Arguments:**

Type	Description
<code>Count_t</code>	Size of this node in pixels (both width and height)

---

Count\_t PixelSizeX()

Query the width of this node (pixels) on the animation display.

**Return Value:**

Type	Description
Count_t	Size of this node in pixels.

---

Count\_t PixelSizeY()

Query the height of this node (pixels) on the animation display.

**Return Value:**

Type	Description
Count_t	Size of this node in pixels.

---

Shape(Shape\_t)

Set the shape for this node on the animation display.

**Arguments:**

Type	Description
Shape_t	Shape for this node. See node.h for shape types.

---

Shape\_t Shape()

Query the shape for this node on the animation display.

**Return Value:**

Type	Description
Shape_t	Shape for this node. See node.h for shape types.

---

CustomShape\_t CustomShape()

Get the custom shape callback function.

**Return Value:**

Type	Description
CustomShape_t	Function pointer to custom shape callback function.

---

CustomShape(CustomShape\_t)

Specify the custom shape callback function.

**Arguments:**

Type	Description
CustomShape_t	Pointer to new custom shape callback function.

---

bool CustomShapeFile(const char\*)

Specifies a file containing the custom shape image. Must be in an image format supported by Qt. NOTE: This **must** be called *after* the call to Simulator::StartAnimation, since it needs a QTWindow object. If called before StartAnimation, will always return false.

**Return Value:**

Type	Description
bool	True if successfully loaded

**Arguments:**

Type	Description
const char*	File name for image.

---

Color(const QColor&)

Specify a color for this node

**Arguments:**

Type	Description
const QColor&	Desired color (see qcolor.h in qt).

---

```
bool HasColor()
```

Query if this node has a color assigned

**Return Value:**

Type	Description
bool	True if color has been specified.

---

```
QColor Color()
```

Query color for this node.

**Return Value:**

Type	Description
QColor	Specified display color for this node.

---

```
NodeAnimation* GetNodeAnimation()
```

Get a pointer to the nodes NodeAnimation information block. Always nil for Ghosts, possibly non-nil for real.

**Return Value:**

Type	Description
NodeAnimation*	Associated NodeAnimation pointer.

---

```
bool ICMPEnabled()
```

Query if ICMP enabled for this node.

**Return Value:**

Type	Description
bool	True if enabled. NOTE. This only means this node will use ICMP if globally enabled (see icmp.h)

---

```
DisableICMP()
```

Specify this node does NOT use ICMP, even if globally enabled.

---

```
Down()
```

Specifies the the node has crashed and should not forward or generate any packets.

---

```
Up()
```

Specifies the node has recovered from failure.

---

```
bool IsDown()
```

Determine if node has failed or not.

**Return Value:**

Type	Description
bool	TRUE if node is down (failed)

---

Node **Static Methods:**

```
SetNextNodeId(NodeId_t)
```

Set a new value for the next node identifier value.

**Arguments:**

Type	Description
NodeId_t	The new next node identifier value.

---

```
const NodeVec_t& GetNodes()
```

Get the vector of all existing nodes.

**Return Value:**

Type	Description
const NodeVec_t&	A const reference to the vector of existing nodes.

---

```
Node* GetNode(NodeId_t)
```

Get the node pointer for the specified node identifier.

**Return Value:**

Type	Description
Node*	A pointer to the specified node. Returns <i>nil</i> if the specified node does not exist.

**Arguments:**

Type	Description
NodeId_t	The node identifier desired.

---

```
Clear()
```

Clear the node vector. Normally only called after the simulation has completed, and we are trying to track memory leaks.

---

```
DefaultPixelSize(Count_t)
```

Set the default size (in pixels) of the animated nodes

**Arguments:**

Type	Description
Count_t	Default size in pixels.

---

```
Count_t DefaultPixelSize()
```

Query the default node pixel size

**Return Value:**

Type	Description
Count_t	The default node pixel size.

---

```
DefaultShape(Shape_t)
```

Set the default shape for nodes on the animation display

**Arguments:**

Type	Description
Shape_t	Default shape. See node.h for shape types.

---

```
Shape_t DefaultShape()
```

Query the default node shape.

**Return Value:**

Type	Description
Shape_t	Default shape. See node.h for shape types.

---

Class `Location`. Contains location information in the X–Y plane for nodes.

`Location` **Constructors:**

```
Location()
```

Default constructor. Sets both X and Y position to zero.

---

Location(Meters\_t, Meters\_t)  
Specifies X and Y locations.

**Arguments:**

Type	Description
Meters_t	X location (meters)
Meters_t	Y location (meters)

---

Location **Public Methods:**

Meters\_t X()  
Get the X location.

**Return Value:**

Type	Description
Meters_t	The X location (meters)

---

## 4.2 Interfaces

Nodes in a *GTNets* simulation have one or more *Interface* objects that model behavior of the hardware interface to a communications link. Interfaces have an associated link object, a layer 2 protocol object, a queue object, a MAC address, an *IP Address* and an address mask. It may often be the case that a *GTNets* simulation will not directly create or refer to *Interface* objects, since the *Node* methods discuss above provide more general methods for creating links between nodes (which in turn creates the interface objects and it's associated helpers). Class *Interface*. Class *interface* is a base class for all interfaces. The various methods for interfaces are documented in the appropriate subclasses.

Interface **Public Methods:**

AddARPEntree(IPAddr\_t, MACAddr)  
Adds a new arp entry to the interface arp cache

**Arguments:**

Type	Description
IPAddr_t	The Ip address of the neighbour
MACAddr	The corresponding Mac address

---

SetIPAddr(IPAddr\_t)  
Specify an *IP Address* associated with this interface.

**Arguments:**

Type	Description
IPAddr_t	The <i>IP Address</i> to assign this interface.

---

IPAddr GetIPAddr()  
Return the *IP Address* associated with this interface.

**Return Value:**

Type	Description
IPAddr	The <i>IP Address</i> associated with this interface. Returns <i>IPADDR_NONE</i> if no <i>IP Address</i> is assigned.

---

Mask\_t GetIPMask()  
Return the *IP Address* mask associated with this interface.

**Return Value:**

Type	Description
Mask_t	The <i>IP Address</i> mask associated with this interface.

---

```
SetMACAddr(const MACAddr&)
```

Set the *MAC* address associated with this interface.

**Arguments:**

Type	Description
const MACAddr&	The <i>MAC</i> address for this interface.

---

```
MACAddr GetMACAddr()
```

Get the *MAC* address associated with this interface.

**Return Value:**

Type	Description
MACAddr	The <i>MAC</i> address for this interface.

---

```
Node* GetNode()
```

Get the node pointer to which this interface is associated.

**Return Value:**

Type	Description
Node*	A pointer to the node this interface is associated with.

---

```
SetNode(Node*)
```

Set the node pointer to which this interface is associated.

**Arguments:**

Type	Description
Node*	Pointer to the node this interface is associated with.

---

```
ScheduleRx(LinkEvent*, Time_t, bool)
```

Schedule a packet receipt event at a future time.

**Arguments:**

Type	Description
LinkEvent*	The link event to add.
Time_t	Simulation time when the receipt will occur.
bool	list. Only false for trace file playbacks. <b>Default Value</b> is true

---

```
EventCQ* GetEventCQ()
```

Get a pointer to the pending RX event circular queue for this interface.

**Return Value:**

Type	Description
EventCQ*	Pointer to EventCQ for this interface.

---

Interface **Static Methods:**

```
DefaultUseARP(bool)
```

Specify that all subsequently created interfaces should use the ARP protocol (or not). Default is NOT.

**Arguments:**

Type	Description
bool	True if use ARP desired on all new interfaces. <b>Default Value</b> is true

---

## 4.3 Queues

Interfaces in *GTNets* have associated queues for buffering packets when a link is busy and a new packet arrives. *GTNets* has queuing models for the standard *DropTail* style queues, and Random Early Detection (RED) queues. The base `Queue` class describes basic functionality required for all queues. Each of these *GTNets* objects are described below:

Class `Queue`. The class `Queue` defines the base class for interface queues that enqueue packets before they are sent down on the link. Different queuing disciplines may inherit from this class to define their own functionality.

### Queue Constructors:

`Queue(bool)`

This method is the default constructor

#### Arguments:

Type	Description
<code>bool</code>	Default Value is true

---

### Queue Public Methods:

`bool Enque(Packet*)`

This method is used to enqueue a packet in the queue

#### Return Value:

Type	Description
<code>bool</code>	a boolean to indicate if the enqueue operation succeeded

#### Arguments:

Type	Description
<code>Packet*</code>	the pointer to the packet to be added to the queue

---

`Packet* Deque()`

This method dequeues a packet from the queue and returns it

#### Return Value:

Type	Description
<code>Packet*</code>	Pointer to the packet

---

`Count_t Length()`

This method returns the size of the queue in terms of bytes

#### Return Value:

Type	Description
<code>Count_t</code>	number of bytes

---

`Count_t LengthPkts()`

This method returns the size of the queue in terms of the number of packets

#### Return Value:

Type	Description
<code>Count_t</code>	number of packets enqueued

---

`Queue* Copy()`

This method makes a copy of the queue and returns a pointer to the copy

#### Return Value:

Type	Description
<code>Queue*</code>	pointer to a copy of the queue

---

```
SetLimit(Count_t)
```

This method sets a limit to the size of the queue. this limit is in terms of bytes

**Arguments:**

Type	Description
Count_t	size of the queue in bytes

---

```
Count_t GetLimit()
```

This method returns the size of the queue in terms of the bytes

**Return Value:**

Type	Description
Count_t	Max limit of the queue size in bytes

---

```
SetLimitPkts(Count_t)
```

This method sets a limit to the size of the queue. this limit is in terms of number of packets that can be enqueued

**Arguments:**

Type	Description
Count_t	size of the queue in packets

---

```
Count_t GetLimitPkts()
```

This method returns the size of the queue in terms of the packets

**Return Value:**

Type	Description
Count_t	Max limit of the queue size in number of packets

---

```
DCount_t Average()
```

This method returns the average queue length

**Return Value:**

Type	Description
DCount_t	average queue length

---

```
ResetAverage()
```

This method resets the averages and starts a new interval to calculate averages

---

```
UpdateAverage()
```

This method updates the averages. ala moving averages

---

```
bool Check(Size_t, Packet*)
```

This method checks if buffer space (of a given size) is available

**Return Value:**

Type	Description
bool	boolean to indicate true or false

**Arguments:**

Type	Description
Size_t	The size to be checked
Packet*	The pointer to a packet to be queued. <b>Default Value</b> is nil

---

```
bool Detailed()
```

This method returns a bool to indicate if detailed queue model is to be used

**Return Value:**

Type	Description
bool	bool to indicate true or false

---

Detailed(bool)

This method is used to set if detailed model is to be used

**Arguments:**

Type	Description
bool	bool to indicate if detailed model is needed

---

Time\_t QueuingDelay()

This method calculates the current queuing delay

**Return Value:**

Type	Description
Time_t	the queuing delay

---

DCount\_t TotalWorkload()

This method calculates the total work load in terms of byte-secs

**Return Value:**

Type	Description
DCount_t	The workload

---

SetInterface(Interface\*)

This method attaches the queue to an interface

**Arguments:**

Type	Description
Interface*	The interface to which it is to be attached

---

Count\_t DropCount()

This method returns the number of dropped packets.

**Return Value:**

Type	Description
Count_t	the number of dropped packets.

---

Count\_t EnqueueCount()

This method returns the number of packets enqueued

**Return Value:**

Type	Description
Count_t	the number of enqueued packets.

---

CountEnq(Packet\*)

**Arguments:**

Type	Description
Packet*	<b>Default Value</b> is nil

---

ResetStats()

This method zeroes down all the statistics pertaining to this queue

---

AddForcedLoss(Time\_t, Count\_t, Time\_t)

Adds information about a forced loss. Allows a simulation to force a packet loss at a particular time, to observe protocol behavior in the presence of losses.

**Arguments:**

Type	Description
Time_t	Time of the loss.
Count_t	Number of consecutive packets to drop, starting at the time. <b>Default Value</b> is 1
Time_t	Expiration time for this loss. If all the requested losses are not accomplished by this time, then the remainder are not enforced. <b>Default Value</b> is INFINITE_TIME

---

```
bool CheckForcedLoss(bool)
```

Checks if a forced loss should be enforced. Called by each subclass prior to enqueuing a packet. The forced loss will be counted, and the entry removed if all forced losses have been enforced.

**Return Value:**

Type	Description
bool	True if time for a forced loss.

**Arguments:**

Type	Description
bool	True if the forced loss entry should be removed by this call. <b>Default Value</b> is true

---

```
bool CheckSpoofedSource(Packet*)
```

Some queue types enforce egress filtering and drop spoofed source addresses. This checks if the packet should be dropped for spoofing. Returns false for all queue types that don't check this, and true for those that do check and that find a spoofed source address.

**Return Value:**

Type	Description
bool	True if packet should be dropped due to spoofing.

**Arguments:**

Type	Description
Packet*	Packet to check.

---

```
EnableTimeSizeLog(bool)
```

Enable or disable the logging of size vs. time history. NOTE. This should be enabled with care, since the memory demands of the log are significant.

**Arguments:**

Type	Description
bool	True if Time/Size logging desired. <b>Default Value</b> is true

---

```
LogSizePackets(bool)
```

Specify that the TimeSize log should log in units of packets (or not). Default is log in bytes.

**Arguments:**

Type	Description
bool	True of logging in units of packets desired. <b>Default Value</b> is true

---

```
PrintTimeSizeLog(std::ostream&, Count_t, char, ')
```

Log previously collected time/size data to a file.

**Arguments:**

Type	Description
std::ostream&	Output stream to log the data. If non-zero, each entry is divided by this value.
Count_t	<b>Default Value</b> is 0
char	Separator character between time and sequence number <b>Default Value</b> is ' '

---

**Queue Static Methods:**

```
Default(const Queue&)
```

This method sets a the default queue

**Arguments:**

Type	Description
const Queue&	reference to a queue object

---

```
Queue* Default()
```

This method returns the default queue object

**Return Value:**

Type	Description
Queue*	Pointer to the default queue

---

```
DefaultLength(Count_t)
```

This method sets the default queue length in bytes

**Arguments:**

Type	Description
Count_t	the count in bytes

---

```
Count_t DefaultLength()
```

This method returns the default queue size in bytes.

**Return Value:**

Type	Description
Count_t	the default size in bytes

---

```
DefaultLimitPkts(Count_t)
```

This method sets the default queue length in terms of the number of packets

**Arguments:**

Type	Description
Count_t	the count in bytes

---

Class DropTail. This method implements the simple droptail or FIFO queue.

DropTail **Constructors:**

```
DropTail()
```

the default constructor

---

```
DropTail(Count_t)
```

This constructor creates a queue of specified size

**Arguments:**

Type	Description
Count_t	the size of the queue in bytes

---

```
DropTail(const DropTail&)
```

This constructor creates a queue identical to another queue

**Arguments:**

Type	Description
const DropTail&	Reference to an existing queue

---

DropTail **Public Methods:**

```
bool Enque(Packet*)
```

This method is used to enqueue a packet in the queue

**Return Value:**

Type	Description
bool	a boolean to indicate if the enqueue operation succeeded or if the packet was dropped.

**Arguments:**

Type	Description
Packet*	the pointer to the packet to be added to the queue

---

```
Packet* Deque()
```

This method dequeues a packet from the queue and returns it

**Return Value:**

Type	Description
Packet*	Pointer to the packet

---

```
Count_t Length()
```

This method returns the size of the queue in terms of bytes

**Return Value:**

Type	Description
Count_t	number of bytes

---

```
Count_t LengthPkts()
```

This method returns the size of the queue in terms of the number of packets

**Return Value:**

Type	Description
Count_t	number of packets enqueued

---

```
SetInterface(Interface*)
```

This method attaches the queue to an interface

**Arguments:**

Type	Description
Interface*	The interface to which it is to be attached

---

```
Queue* Copy()
```

This method makes a copy of the queue and returns a pointer to the copy

**Return Value:**

Type	Description
Queue*	pointer to a copy of the queue

---

```
SetLimit(Count_t)
```

This method sets a limit to the size of the queue. this limit is in terms of bytes

**Arguments:**

Type	Description
Count_t	size of the queue in bytes

---

```
Count_t GetLimit()
```

This method returns the size of the queue in terms of the bytes

**Return Value:**

Type	Description
Count_t	Max limit of the queue size in bytes

---

```
SetLimitPkts(Count_t)
```

This method sets a limit to the size of the queue. this limit is in terms of number of packets that can be enqueued

**Arguments:**

Type	Description
Count_t	size of the queue in packets

---

```
Count_t GetLimitPkts()
```

This method returns the size of the queue in terms of the packets

**Return Value:**

Type	Description
Count_t	Max limit of the queue size in number of packets

---

```
Time_t QueuingDelay()
```

This method calculates the current queuing delay

**Return Value:**

Type	Description
Time_t	the queuing delay

---

```
bool Check(Size_t, Packet*)
```

This method checks if buffer space (of a given size) is available

**Return Value:**

Type	Description
bool	boolean to indicate true or false

**Arguments:**

Type	Description
Size_t	The size to be checked
Packet*	The pointer to a packet to be queued. <b>Default Value</b> is nil

---

```
Count_t Limit()
```

This method returns the maximum limit of the queue size.

**Return Value:**

Type	Description
Count_t	the limit (in bytes)

---

Class REDQueue. This class derives from the class DropTail and implements the Random Early Detection Queuing discipline. More references about RED can be found at Sally Floyd's webpages

REDQueue **Constructors:**

```
REDQueue()
```

Default constructor

---

```
REDQueue(DCount_t, Count_t, Count_t, Count_t, DCount_t, Count_t)
```

This constructor the critical RED parameters and builds a corresponding RED queue

**Arguments:**

Type	Description
DCount_t	weight of the queue
Count_t	minimum threshold
Count_t	maximum threshold
Count_t	Limit/max size for the queue
DCount_t	maximum value for mark/drop probability
Count_t	Average packet size

---

REDQueue **Public Methods:**

```
REDQueue()
```

Destructor

---

```
bool Enque(Packet*)
```

This method is used to enqueue a packet in the queue

**Return Value:**

Type	Description
bool	a boolean to indicate if the enqueue operation succeeded

**Arguments:**

Type	Description
Packet*	the pointer to the packet to be added to the queue

---

```
Packet* Deque()
```

This method dequeues a packet from the queue and returns it

**Return Value:**

Type	Description
Packet*	Pointer to the packet

---

```
Packet* MarkPacketwProb()
```

This method marks a packet with its mark/drop probability. This method is used when the average queue is still between the min and max thresholds

**Return Value:**

Type	Description
Packet*	the marked packet

---

```
Packet* MarkPacket()
```

This method marks a packet and returns a pointer to it. This method is called when the current queue size is larger than the max threshold.

**Return Value:**

Type	Description
Packet*	Pointer to the packet

---

```
Packet* DropPacket()
```

This method drops the packet from the queue, it is called when we need an unconditional drop. like when the queue overflows

---

```
Count_t Length()
```

This method returns the size of the queue in terms of bytes

**Return Value:**

Type	Description
Count_t	number of bytes

---

```
Queue* Copy()
```

This method makes a copy of the queue and returns a pointer to the copy

**Return Value:**

Type	Description
Queue*	pointer to a copy of the queue

---

```
SetInterface(Interface*)
```

This method attaches the queue to an interface

**Arguments:**

Type	Description
Interface*	The interface to which it is to be attached

---

```
Rate_t Bandwidth()
```

This method returns the bandwidth of the link associated with the interface that this queue is attached to.

**Return Value:**

Type	Description
Rate_t	the bandwidth of the link

---

## 4.4 Links

Node objects in *GTNets* are connected with `Link` objects. `Link` objects can be either point-to-point, LAN's (an Ethernet object), or wireless. All link objects are subclassed from the basic `Link` object, which describes the functionality required for all links. The base class `Link` as well as the various link subclasses defined in *GTNets* is described below.

**Class `Link`.** Class `Link` is a virtual base class, describing functionality needed by all *GTNets* links. Since there are several pure virtual functions, no objects of class `Link` can be created. The *GTNets* objects `Linkp2p`, `Ethernet` and `WirelessLink` are subclasses and provide the specific functionality of those types of links.

**Link Constructors:**

```
Link()
```

Creates a link with the default bandwidth and delay.

---

```
Link(Rate_t, Time_t)
```

Creates a link with the default bandwidth and delay.

**Arguments:**

Type	Description
Rate_t	Bandwidth for the link.
Time_t	Propagation delay for the link

---

**Link Public Methods:**

```
Link* Copy()
```

Make a copy of this `Link` object.

**Return Value:**

Type	Description
Link*	A pointer to the copied object.

---

```
Interface* GetPeer(Packet*)
```

Get the interface pointer for the remote endpoint of this link for the specified packet.

**Return Value:**

Type	Description
Interface*	A pointer to the receiving interface for the specified packet.

**Arguments:**

Type	Description
Packet*	The packet being transmitted.

---

```
Interface* GetPeer(Count_t)
```

Get the interface pointer for the remote endpoint of this link for the specified peer index

**Return Value:**

Type	Description
Interface*	A pointer to the receiving interface for the specified peer

**Arguments:**

Type	Description
Count_t	Which peer desired (0 < i < NeighborCount())

---

```
bool Transmit(Packet*, Interface*, Node*)
```

Transmit a packet.

**Return Value:**

Type	Description
------	-------------

bool	True if the transmit succeeded, false if packet was dropped.
------	--

**Arguments:**

Type	Description
------	-------------

Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.

---

```
bool Transmit(Packet*, Interface*, Node*, Rate_t)
```

Transmit a packet.

**Return Value:**

Type	Description
------	-------------

bool	True if the transmit succeeded, false if packet was dropped.
------	--

**Arguments:**

Type	Description
------	-------------

Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.
Rate_t	The rate at which the packet is transmitted.

---

```
bool Transmit(LinkEvent*, Interface*, Time_t)
```

Transmit using an existing LinkEvent pointer. Used by satellite nodes.

**Arguments:**

Type	Description
------	-------------

LinkEvent*	Existing LinkEvent pointer
Interface*	Receiving Interface pointer
Time_t	Receipt time

---

```
Handle(Event*, Time_t)
```

Link objects are subclass of Handler and must define a Handle method to handle events scheduled for this link.

**Arguments:**

Type	Description
------	-------------

Event*	The event to handle.
Time_t	The current simulation time.

---

```
Bandwidth(Rate_t)
```

Set the link bandwidth to the specified value.

**Arguments:**

Type	Description
------	-------------

Rate_t	Link bandwidth to set.
--------	------------------------

---

```
Rate_t Bandwidth()
```

Return the bandwidth for this link.

**Return Value:**

Type	Description
------	-------------

Rate_t	The bandwidth for the link.
--------	-----------------------------

---

```
bool Busy()
```

Determine if the link is currently busy transmitting a packet.

**Return Value:**

Type	Description
bool	True if link busy, false if not.

---

```
Count_t PeerCount()
```

Returns the number of peers (nodes) that this link provides connectivity to. This method must be defined by all subclasses.

**Return Value:**

Type	Description
Count_t	Count of peers for this link.

---

```
IPAddr_t PeerIP(Count_t)
```

Get the *IP Address* of the specified peer. This method must be defined by all subclasses.

**Return Value:**

Type	Description
IPAddr_t	<i>IP Address</i> of the requested peer. Returns IPADDR_NONE if peer does not exist.

**Arguments:**

Type	Description
Count_t	Index of the peer desired, in the range [0..PeerCount).

---

```
IPAddr_t NodePeerIP(Node*)
```

Get the *IP Address* of the specified peer. This method must be defined by all subclasses.

**Return Value:**

Type	Description
IPAddr_t	<i>IP Address</i> of the requested peer. Returns IPADDR_NONE if specified node is not a peer.

**Arguments:**

Type	Description
Node*	Pointer to desired peer node.

---

```
bool NodeIsPeer(Node*)
```

Determine if specified node is a peer. This method must be defined by all subclasses.

**Return Value:**

Type	Description
bool	True if node is a peer, false if not.

**Arguments:**

Type	Description
Node*	Pointer to desired peer node.

---

```
Count_t NodePeerIndex(Node*)
```

Get the peer index, in the range [0..PeerCount) for the specified node.

**Return Value:**

Type	Description
Count_t	Peer index for specified node.

**Arguments:**

Type	Description
Node*	Pointer to desired peer node.

---

```
IPAddr_t DefaultPeer(Interface*)
```

Find the *IP Address* of the default peer. Link classes with multiple peers (such as an Ethernet link) can specify one of the peers as the *Default Gateway*. This simplifies routing decisions in some cases. This method must be defined by all subclasses.

**Return Value:**

Type	Description
IPAddr_t	<i>IP Address</i> of the default peer. Returns IPADDR_NONE if no default peer exists.

**Arguments:**

Type	Description
Interface*	Interface pointer for the interface of this link.

---

```
BitErrorRate(Mult_t)
```

Specify a bit error rate for this link.

**Arguments:**

Type	Description
Mult_t	The desired bit error rate.

---

```
Mult_t BitErrorRate()
```

Return the bit error rate for this link.

**Return Value:**

Type	Description
Mult_t	The bit error rate assigned to this link.

---

```
ResetUtilization()
```

Reset the link utilization statistics for this link to zero.

---

```
Mult_t Utilization()
```

Return the utilization for this link, in the range [0..1].

**Return Value:**

Type	Description
Mult_t	The link utilization for this link.

---

```
AddNotify(NotifyHandler*, void*)
```

Higher layer protocol entities can request a notification when links go non-busy. This method adds a notification handler to the list.

**Arguments:**

Type	Description
NotifyHandler*	Any object subclassed from <code>NotifyHandler</code> . The <code>Notify</code> method for that handler will be called when this link goes non-busy.
void*	A blind pointer to be passed to the <code>Notify</code> method.

---

```
Weight_t Weight()
```

Query the link weight (used by some routing protocols) for this link.

**Return Value:**

Type	Description
Weight_t	The weight for this link.

---

```
Weight(Weight_t)
```

Assign a weight (used by some routing protocols) to this link.

**Arguments:**

Type	Description
Weight_t	Weight to assign to this link.

---

`Jitter(const Random&)`

Specify a random number generator object used for *jitter*. Jitter is used to randomly delay the packet transmission time by small perturbations, to model CPU processing time in routers. No jitter value is used unless this method is used.

**Arguments:**

Type	Description
<code>const Random&amp;</code>	Any random number generator object. The object is copied.

---

`Time_t NextFreeTime()`

Predict the time this link will be free. This is useful only for links without jitter, and without contention.

**Return Value:**

Type	Description
<code>Time_t</code>	The predicted time this link will be free.

---

`Count_t NeighborCount(Node*)`

Return the number of routing neighbors on this link.

**Return Value:**

Type	Description
<code>Count_t</code>	Count of routing neighbors.

**Arguments:**

Type	Description
<code>Node*</code>	Node querying neighbors (significant for Ethernet links)

---

`Neighbors(NodeWeightVec_t&, Interface*, bool)`

Get a vector of all routing neighbors.

**Arguments:**

Type	Description
<code>NodeWeightVec_t&amp;</code>	Returned vector of routing neighbors.
<code>Interface*</code>	Interface pointer to associated interface.
<code>bool</code>	True if include all neighbors, even for single gateway Ethernet.

---

`AllNeighbors(NodeWeightVec_t&)`

Find all neighbors for this link, even if they are not routing neighbors. Used by broadcasts to find all receivers.

**Arguments:**

Type	Description
<code>NodeWeightVec_t&amp;</code>	Returned vector of neighbors.

---

`DCount_t BytesSent()`

Get a count of the total number of bytes sent on this link.

**Return Value:**

Type	Description
<code>DCount_t</code>	Count of total bytes sent on this link.

---

`DCount_t PktsSent()`

Get a count of the total number of packets sent on this link.

**Return Value:**

Type	Description
<code>DCount_t</code>	Count of total packets sent on this link.

---

MACAddr IPToMac(IPAddr\_t)  
Convert a peer *IP Address* to peer MAC address.

**Return Value:**

Type	Description
MACAddr	MAC address for that peer.

**Arguments:**

Type	Description
IPAddr_t	<i>IP Address</i> of desired peer.

---

bool RxBroadcast()  
Query if this link should receive a copy of its own broadcasts.

**Return Value:**

Type	Description
bool	True if receive own broadcast, false if not.

---

**Link Static Methods:**

DefaultRate(Rate\_t)  
Set a default transmission rate to be used for any new link created.

**Arguments:**

Type	Description
Rate_t	Desired default transmission rate (bits/second)

---

Rate\_t DefaultRate()  
Return the default transmission rate.

**Return Value:**

Type	Description
Rate_t	The default transmission rate (bits/second).

---

DefaultDelay(Time\_t)  
Set a default propagation delay to be used for any new link created.

**Arguments:**

Type	Description
Time_t	Desired default propagation delay (second)

---

Time\_t DefaultDelay()  
Return the default propagation delay

**Return Value:**

Type	Description
Time_t	The default propagation delay (second).

---

DefaultJitter(const Random&)  
Set a default random variable for link transmission jitter.

**Arguments:**

Type	Description
const Random&	Any random number generator object.

---

Random\* DefaultJitter()  
Get a pointer to the default jitter random number generator.

**Return Value:**

Type	Description
Random*	Pointer to the default jitter random number generator.

---

Default(const Link&)

Set a default link object to use when a link object is needed and none has been specified.

**Arguments:**

Type	Description
const Link&	Link object to use for the default link.

---

const Link& Default()

Get a reference to the default link object.

**Return Value:**

Type	Description
const Link&	A constant reference to the default link object.

---

SeedBER(Seed\_t)

Seed the bit error rate random number generator

**Arguments:**

Type	Description
Seed_t	A random seed value for te bit error rate random number generator.

---

UseSeqEvents(bool)

A small performance improvement can be gained by leveraging the fact that link receive events are always in strictly increasing timestamp order for a single link. Setting UseSeqEvents to true enables this optimization. NOTE. The GTNetS packet animation code depends on this setting to find which packets are in-flight on the links. If this setting is FALSE, no packets will be displayed on the animation display.

**Arguments:**

Type	Description
bool	True if using sequential events optimization, false if not.

---

Class Linkp2p. Class Linkp2p defines a serial point to point link object derived from the Link object

Linkp2p **Public Methods:**

Link\* Copy()

This method makes a copy of itself and returns a pointer to the copy

**Return Value:**

Type	Description
Link*	Pointer to the new Link object

---

SetPeer(Interface\*)

This method sets a specific interface as the peer

**Arguments:**

Type	Description
Interface*	The interface to be set as the peer interface

---

Interface\* GetPeer(Packet\*)

This method looks at the packet's destination MAC address and returns the interface corresponding to it

**Return Value:**

Type	Description
Interface*	The destination interface

**Arguments:**

Type	Description
Packet*	The packet to be transmitted

---

```
Interface* GetPeer(Count_t)
```

Get the interface pointer for the remote endpoint of this link for the specified peer index.

**Return Value:**

Type	Description
Interface*	A pointer to the receiving interface for the specified peer

**Arguments:**

Type	Description
Count_t	Which peer desired (0 < i < NeighborCount())

---

```
bool Transmit(Packet*, Interface*, Node*)
```

Transmit a packet.

**Return Value:**

Type	Description
bool	True if the transmit succeeded, false if packet was dropped.

**Arguments:**

Type	Description
Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.

---

```
bool Transmit(Packet*, Interface*, Node*, Rate_t)
```

Transmit a packet.

**Return Value:**

Type	Description
bool	True if the transmit succeeded, false if packet was dropped.

**Arguments:**

Type	Description
Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.
Rate_t	Transmitting rate (bits/sec)

---

```
MACAddr IPToMac(IPAddr_t)
```

Convert peer IP to peer MAC

**Return Value:**

Type	Description
MACAddr	The corresponding MAC address

**Arguments:**

Type	Description
IPAddr_t	The IP Address whose MAC is to be determined

---

```
Count_t NeighborCount(Node*)
```

Returns the number of neighbors on this link which are routers technically, count of routing peers

**Return Value:**

Type	Description
Count_t	The number of routing peers.

**Arguments:**

Type	Description
Node*	Node querying neighbors (significant for Ethernet links)

---

```
Neighbors(NodeWeightVec_t&, Interface*, bool)
```

Build the vector of neighbor,interface list

**Arguments:**

Type	Description
NodeWeightVec_t&	The NodeWightVec element where we will fill in the neighbors
Interface*	The interface, with respect to which we need to find these neighbors
bool	boolean to specify if we need all the neighbors OR if we need only routing peers

---

```
AllNeighbors(NodeWeightVec_t&)
```

AllNeighbors is used by broadcasts even if they are not routing peers. Use this to find all the peers

**Arguments:**

Type	Description
NodeWeightVec_t&	The NodeWeightVec element, with respect to which, we need all the peers in this link

---

```
Count_t PeerCount()
```

Returns the number of peers on this link

**Return Value:**

Type	Description
Count_t	number of peers

---

```
IPAddr_t PeerIP(Count_t)
```

Since each node on this link is also associated with a count this returns the IP address of the n'th peer

**Return Value:**

Type	Description
IPAddr_t	the corresopnding IPAddress

**Arguments:**

Type	Description
Count_t	the count index n

---

```
IPAddr_t NodePeerIP(Node*)
```

Return the IP address of the node if it is on this link and is found

**Return Value:**

Type	Description
IPAddr_t	The IP Address

**Arguments:**

Type	Description
Node*	The pointer to the node object

---

```
bool NodeIsPeer(Node*)
```

Find out if node is a peer

**Return Value:**

Type	Description
bool	boolean result of the test

**Arguments:**

Type	Description
Node*	The Node

---

```
Count_t NodePeerIndex(Node*)
```

Returns the index of the node in this link

**Return Value:**

Type	Description
Count_t	The index of the peer

**Arguments:**

Type	Description
Node*	The pointer to the node

---

```
IPAddr_t DefaultPeer(Interface*)
```

Find the default peer's IP Address; this will find the default gateway's IP Address in the same subnet

**Return Value:**

Type	Description
IPAddr_t	The default peer's IP

**Arguments:**

Type	Description
Interface*	The interface whose default peerIP is to be known

---

Linkp2p **Static Methods:**

```
Linkp2p& Default()
```

Returns the default link

**Return Value:**

Type	Description
Linkp2p&	a reference to the default point-to-point link

---

```
Linkp2p& Default(const Linkp2p&)
```

Sets a new default linkp2p object, and returns

**Return Value:**

Type	Description
Linkp2p&	a reference to the new default point-to-point link

**Arguments:**

Type	Description
const Linkp2p&	a reference to the new default linkp2p object

---

Class `Ethernet`. Class `Ethernet` defines the Ethernet Link object. It is derived from the `Link` object. The ethernet object creates an ethernet with a user-defined number of nodes.

`Ethernet` **Constructors:**

```
Ethernet(Count_t, IPAddr_t, Mask_t, SystemId_t, Rate_t, Time_t, Detail_t)
```

The constructor for the `Ethernet` object that creates an an ethernet, with a given number of nodes and the associated interfaces. This constructor assumes that we create a subnet without any default gateway.

**Arguments:**

Type	Description
Count_t	The number of nodes in this ethernet
IPAddr_t	The beginning IP address of this ethernet
Mask_t	The subnet mask to be used for this ethernet.
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0
Rate_t	The Transmission Rate of the link in bps. (may use <code>ratetimeparse</code> for simplicity) <b>Default Value</b> is <code>Rate"10Mb"</code>
Time_t	The link delay in secs. (may use <code>ratetimeparse</code> for simplicity) <b>Default Value</b> is <code>Time"1us"</code>
Detail_t	A detail type whether <code>PARTIAL</code> , <code>FULL</code> or <code>NONE</code> <b>Default Value</b> is <code>PARTIAL</code>

---

```
Ethernet(Count_t, IPAddr_t, Mask_t, const L4Protocol&, SystemId_t, Rate_t,
Time_t, Detail_t)
```

The constructor for the Ethernet object that creates an an ethernet, with a given number of nodes and the associated interfaces. This constructor assumes that we create a subnet without any default gateway. This one also binds the specified l4 protocol to each node (presumably a TCP Server)

**Arguments:**

Type	Description
Count_t	The number of nodes in this ethernet
IPAddr_t	The beginning IP address of this ethernet
Mask_t	The subnet mask to be used for this ethernet.
const L4Protocol&	The layer 4 protocol to bind to each node
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0
Rate_t	The Transmission Rate of the link in bps. (may use ratetimeparse for simplicity) <b>Default Value</b> is Rate"10Mb"
Time_t	The link delay in secs. (may use ratetimeparse for simplicity) <b>Default Value</b> is Time"1us"
Detail_t	A detail type whether PARTIAL, FULL or NONE <b>Default Value</b> is PARTIAL

---

```
Ethernet(Count_t, Node*, IPAddr_t, Mask_t, SystemId_t, Rate_t, Time_t, Detail_t)
```

This constructor creates an Ethernet object that has a gateway node and all of the rest of the nodes use this as a default route. The gateway node is assumed to already exist.

**Arguments:**

Type	Description
Count_t	The number of nodes in this ethernet
Node*	Pointer to the gateway node object
IPAddr_t	The beginning IP address of this ethernet
Mask_t	The subnet mask to be used for this ethernet.
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0
Rate_t	The Transmission Rate of the link in bps. (may use ratetimeparse for simplicity) <b>Default Value</b> is Rate"10Mb"
Time_t	The link delay in secs. (may use ratetimeparse for simplicity) <b>Default Value</b> is Time"1us"
Detail_t	A detail type whether PARTIAL, FULL or NONE <b>Default Value</b> is PARTIAL

---

```
Ethernet(Count_t, Node*, IPAddr_t, Mask_t, const L4Protocol&, SystemId_t,
Rate_t, Time_t, Detail_t)
```

This constructor creates an `Ethernet` object that has a gateway node and all of the rest of the nodes use this as a default route. The gateway node is assumed to already exist. Also specifies a layer 4 protocol object to bind at each node.

**Arguments:**

Type	Description
Count_t	The number of nodes in this ethernet
Node*	Pointer to the gateway node object
IPAddr_t	The beginning IP address of this ethernet
Mask_t	The subnet mask to be used for this ethernet.
const L4Protocol&	The layer 4 protocol to bind to each node
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0
Rate_t	The Transmission Rate of the link in bps. (may use <code>ratetimeparse</code> for simplicity) <b>Default Value</b> is <code>Rate"10Mb"</code>
Time_t	The link delay in secs. (may use <code>ratetimeparse</code> for simplicity) <b>Default Value</b> is <code>Time"1us"</code>
Detail_t	A detail type whether PARTIAL, FULL or NONE <b>Default Value</b> is PARTIAL

---

**Ethernet Public Methods:**

```
Link* Copy()
This is a copy constructor
```

---

```
Interface* GetPeer(Packet*)
This method returns the destination interface by looking at the destination MAC address of the packet.
```

**Return Value:**

Type	Description
Interface*	The destination interface.

**Arguments:**

Type	Description
Packet*	The packet to be transmitted.

---

```
Interface* GetPeer(Count_t)
Get the interface pointer for the remote endpoint of this link for the specified peer index
```

**Return Value:**

Type	Description
Interface*	A pointer to the receiving interface for the specified peer

**Arguments:**

Type	Description
Count_t	Which peer desired ( $0 < i < \text{NeighborCount}()$ )

---

```
bool Transmit(Packet*, Interface*, Node*)
Transmit a packet.
```

**Return Value:**

Type	Description
bool	True if the transmit succeeded, false if packet was dropped.

**Arguments:**

Type	Description
Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.

---

```
Count_t PeerCount()
```

Returns the number of peers on this link

**Return Value:**

Type	Description
Count_t	number of peers

---

```
IPAddr_t PeerIP(Count_t)
```

Since each node on this link is also associated with a count this returns the IP address of the n'th peer

**Return Value:**

Type	Description
IPAddr_t	the corresponding IP Address

**Arguments:**

Type	Description
Count_t	the count index n

---

```
IPAddr_t NodePeerIP(Node*)
```

Return the IP address of the node if it is on this link and is found

**Return Value:**

Type	Description
IPAddr_t	The IP Address

**Arguments:**

Type	Description
Node*	The pointer to the node object

---

```
bool NodeIsPeer(Node*)
```

Find out if node is a peer

**Return Value:**

Type	Description
bool	boolean result of the test

**Arguments:**

Type	Description
Node*	The Node

---

```
Count_t NodePeerIndex(Node*)
```

Returns the index of the node in this link

**Return Value:**

Type	Description
Count_t	The index of the peer

**Arguments:**

Type	Description
Node*	The pointer to the node

---

```
IPAddr_t DefaultPeer(Interface*)
```

Find the default peer's IP Address, this will find the default gateway's IP Address in the same subnet

**Return Value:**

Type	Description
IPAddr_t	The default peer's IP

**Arguments:**

Type	Description
Interface*	The interface whose default peerIP is to be known

---

```
Count_t NeighborCount(Node*)
```

Returns the number of neighbors on this link which are routers technically, count of routing peers

**Return Value:**

Type	Description
Count_t	The number of routing peers.

**Arguments:**

Type	Description
Node*	Node querying neighbors (significant for Ethernet links)

---

```
Neighbors(NodeWeightVec_t&, Interface*, bool)
```

Build the vector of neighbor,interface list

**Arguments:**

Type	Description
NodeWeightVec_t&	The NodeWightVec element where we will fill in the neighbors
Interface*	The interface, with respect to which we need to find these neighbors
bool	boolean to specify if we need all the neighbors OR if we need only routing peers

---

```
AllNeighbors(NodeWeightVec_t&)
```

AllNeighbors is used by broadcasts even if they are not routing peers. Use this to find all the peers

**Arguments:**

Type	Description
NodeWeightVec_t&	The NodeWeightVec element, with respect to which, we need all the peers in this link

---

```
MACAddr IPToMac(IPAddr_t)
```

Convert peer IP to peer MAC

**Return Value:**

Type	Description
MACAddr	The corresponding MAC address

**Arguments:**

Type	Description
IPAddr_t	The IP Address whose MAC is to be determined

---

```
bool RxBroadcast()
```

---

```
Addnode(Node*)
```

Add a new node to the ethernet

**Arguments:**

Type	Description
Node*	The pointer to the node to be added

---

```
Node* GetNode(Count_t)
```

Get the node by its index

**Return Value:**

Type	Description
Node*	Pointer to the node

**Arguments:**

Type	Description
Count_t	The index to the list of nodes

---

```
Interface* GetIf(Count_t)
```

Get the interface of the node connected to this link by its index

**Return Value:**

Type	Description
Interface*	the interface of the node that attaches to this link

**Arguments:**

Type	Description
Count_t	the index of the node

---

```
IPAddr_t GetIP(Count_t)
```

Get IP of the node by the index of the node

**Return Value:**

Type	Description
IPAddr_t	IP address if the index is within range, else IPADDR_NONE

**Arguments:**

Type	Description
Count_t	int index of the node

---

```
Interface* GetIfByMac(MACAddr)
```

Get a pointer to the interface by its MAC address

**Return Value:**

Type	Description
Interface*	pointer to the interface

**Arguments:**

Type	Description
MACAddr	the MAC address

---

```
Node* Gateway()
```

Returns the gateway node of this LAN

**Return Value:**

Type	Description
Node*	Pointer to the gateway node

---

```
SetLeafQLimit(Count_t)
```

Method to set the queue limits of all the nodes in this link to a specified value

**Arguments:**

Type	Description
Count_t	the size of each queue

---

```
RxOwnBroadcast(bool)
```

Checks whether nodes receive their own broadcasts on this link

**Return Value:**

Type	Description
void	Return true if node receive their own broadcasts; false otherwise

**Arguments:**

Type	Description
bool	Default Value is true

---

Class `WirelessLink`. Class `WirelessLink` is a subclass derived from class `Link` and it describes a typical wireless link. It currently makes some assumptions which are valid only in the case of a wireless LAN but may be extended to any wireless physical link.

**WirelessLink Constructors:**

WirelessLink(Count\_t, IPAddr\_t, Mask\_t, L4Protocol\*, Rate\_t, Time\_t, Detail\_t, bool)

This is the default constructor for this link. It constructs a default WirelessLink of a given number of nodes.

**Arguments:**

Type	Description
Count_t	Number of nodes in the LAN
IPAddr_t	The IP Address of the first node in a given mask
Mask_t	The subnet mask
L4Protocol*	The layer 4 protocol to bind to each node. Currently just nil <b>Default Value</b> is nil
Rate_t	The Rate of the link. <b>Default Value</b> is Rate"11Mb"
Time_t	The delay in the link. <b>Default Value</b> is Time"1us"
Detail_t	The level of detail we want to model. Currently there is no difference in the model based on this argument. <b>Default Value</b> is PARTIAL
bool	<b>Default Value</b> is false

**WirelessLink Public Methods:**

Link\* Copy()

This method is used to make rigid copies of the object

**Return Value:**

Type	Description
Link*	A pointer to the newly created copy

bool Transmit(Packet\*, Interface\*, Node\*)

This method is used to transmit the Packet from a give node

**Arguments:**

Type	Description
Packet*	The packet to be transmitted
Interface*	The interface from which the packet is transmitted
Node*	The node from which the packet is transmitted

bool Transmit(Packet\*, Interface\*, Node\*, Rate\_t)

Transmit a packet.

**Arguments:**

Type	Description
Packet*	The packet to transmit.
Interface*	A pointer to the local (transmitting) interface
Node*	A pointer to the local (transmitting) node.
Rate_t	The rate at which the packet is transmitted.

Time\_t SpeedOfLight(Node\*, Node\*)

This method returns the speed of light delay between the two nodes.

**Arguments:**

Type	Description
Node*	The first node
Node*	

Count\_t PeerCount()

Returns the number of peers on this link

**Return Value:**

Type	Description
Count_t	number of peers

---

```
Count_t NodeCount()
```

Returns the total number of nodes on this link

**Return Value:**

Type	Description
Count_t	Node Count;

---

```
IPAddr_t PeerIP(long, int)
```

Since each node on this link is also associated with a count this returns the IP address of the n'th peer

**Return Value:**

Type	Description
IPAddr_t	the corresopnding IPAddress

**Arguments:**

Type	Description
long	the count index n
int	

---

```
IPAddr_t NodePeerIP(Node*)
```

Return the IP address of the node if it is on this link and is found

**Return Value:**

Type	Description
IPAddr_t	The IP Address

**Arguments:**

Type	Description
Node*	The pointer to the node object

---

```
IPAddr_t DefaultPeer(Interface*)
```

Find the default peer's IP Address, this will find the default gateway's IP Address in the same subnet This method is valid only if we have a staic wireless LAN

**Return Value:**

Type	Description
IPAddr_t	The default peer's IP

**Arguments:**

Type	Description
Interface*	The interface whose default peerIP is to be known

---

```
AllNeighbors(NodeWeightVec_t, nw)
```

AllNeighbors is used by broadcasts even if they are not routing peers. Use this to find all the peers

**Arguments:**

Type	Description
NodeWeightVec_t	The NodeWeightVec element, with respect to which, we need all the peers in this link
nw	

---

```
Neighbors(NodeWeightVec_t, Interface, bool)
```

AllNeighbors is used by broadcasts even if they are not routing peers. Use this to find all the peers

**Arguments:**

Type	Description
NodeWeightVec_t	The NodeWeightVec element, with respect to which, we need all the peers in this link
Interface	
bool	

---

```
bool NodeIsPeer(Node*)
```

Find out if node is a peer

**Return Value:**

Type	Description
bool	boolean result of the test

**Arguments:**

Type	Description
Node*	The Node

---

```
Count_t NodePeerIndex(Node*)
```

Returns the index of the node in this link

**Return Value:**

Type	Description
Count_t	The index of the peer

**Arguments:**

Type	Description
Node*	The pointer to the node

---

```
Count_t NeighborCount(Node*)
```

Returns the number of neighbors on this link which are routers technically, count of routing peers

**Return Value:**

Type	Description
Count_t	The number of routing peers.

**Arguments:**

Type	Description
Node*	Node querying neighbors (significant for Ethernet links)

---

```
Interface* GetIfByMac(MACAddr)
```

Get a pointer to the interface by its MAC address

**Return Value:**

Type	Description
Interface*	pointer to the interface

**Arguments:**

Type	Description
MACAddr	the MAC address

---

```
MACAddr IPToMac(IPAddr_t)
```

Convert peer IP to peer MAC

**Return Value:**

Type	Description
MACAddr	The corresponding MAC address

**Arguments:**

Type	Description
IPAddr_t	The IP Address whose MAC is to be determined

---

```
ReserveLink(void)
```

This method reserves the link for a given transmission interval

**Arguments:**

Type	Description
void	

---

```
FreeLink(void)
```

This method frees the link into IDLE state after a transmission

**Arguments:**

Type	Description
void	

---

```
Interface* GetPeer(Packet*)
```

This method returns the destination interface by looking at the destination MAC address of the packet.

**Return Value:**

Type	Description
Interface*	The destination interface.

**Arguments:**

Type	Description
Packet*	The packet to be transmitted.

---

```
Interface* GetPeer(Count_t)
```

Get the interface pointer for the remote endpoint of this link for the specified peer index

**Return Value:**

Type	Description
Interface*	A pointer to the receiving interface for the specified peer

**Arguments:**

Type	Description
Count_t	Which peer desired (0 <- i < NeighborCount())

---

```
bool IsIdle()
```

This method returns a bool to indicate whether a link is idle/busy

**Return Value:**

Type	Description
bool	true if the link is idle, false otherwise

---

```
IPAddr_t GetIP(Count_t)
```

Get IP of the node by the index of the node

**Return Value:**

Type	Description
IPAddr_t	IP address if the index is within range, else IPADDR_NONE

**Arguments:**

Type	Description
Count_t	int index of the node

---

```
Node* GetNode(Count_t)
```

Get the node by its index

**Return Value:**

Type	Description
Node*	Pointer to the node

**Arguments:**

Type	Description
Count_t	The index to the list of nodes

---

```
Interface* GetIf(Count_t)
```

Get the interface of the node connected to this link by its index

**Return Value:**

Type	Description
Interface*	the interface of the node that attaches to this link

**Arguments:**

Type	Description
Count_t	the index of the node

---

## 4.5 Stock Objects

GTNets includes a number of *Stock Objects* that allow for easy creating of commonly used topologies. These are

- Star topology, with a single node in the center and point-to-point links to a number of *leaf* nodes.
- Grid topology, with nodes positioned in a regular 2 dimensional grid and interconnecting point-to-point links.
- Dumbbell topology, with a single bottleneck link connecting a number of nodes on either side.
- Tree topology, with a single root node, a specified number of levels, and a fanout value specifying number of children per level.
- Random topology using the *Georgia Tech Internet Topology Modeler (GTITM)*.

Each of these are described below.

### 4.5.1 GTitm Modeller

Class GTitm. The class GTitm defines an interface to the Georgia Tech Internetwork Topology Models (GT-ITM) Given the number of transit nodes, subdomains, subnodes, the topology is generated, and accessed through this object handler

**GTitm Constructors:**

```
GTitm(int, int, int, int)
```

This method is the default constructor

**Arguments:**

Type	Description
int	Number of stubdomains on average per transit node
int	Number of transit nodes
int	Number of nodes on average per stubdomain
int	Number of leaf nodes per node

---

**GTitm Public Methods:**

```
TNodeList_t GetTList()
```

This method is used to return the list of all Transit nodes

**Return Value:**

Type	Description
TNodeList_t	A vector of transit nodes

---

```
SubList_t GetSList()
```

This method is used to return the list of all SubDomains

**Return Value:**

Type	Description
SubList_t	A vector of subdomains

---

```
SubList_t GTitm::GetSList(int)
```

This method is used to return the list of SubDomains associated with the given Transit node

**Return Value:**

Type	Description
SubList_t	A vector of subdomains

**Arguments:**

Type	Description
int	Transit node number

---

```
NodeList_t GTitm::GetNList(int)
```

This method is used to return the list of Nodes in the subdomain with id of "subNum"

**Return Value:**

Type	Description
NodeList_t	A vector of nodes

**Arguments:**

Type	Description
int	Subdomain number

---

```
Node* GTitm::GetTransNode(int)
```

This method is used to return a transit node

**Return Value:**

Type	Description
Node*	A pointer to the transit node

**Arguments:**

Type	Description
int	Transit node number

---

```
Node* GTitm::GetSubNode(int, int)
```

This method is used to return a node given a Tnode id and the node id

**Return Value:**

Type	Description
Node*	A pointer to a subnode (node)

**Arguments:**

Type	Description
int	Transit node number
int	Node number

---

## 4.5.2 TransitNode

Class `TransitNode`. Defines an interface for the transit node

**TransitNode Constructors:**

```
TransitNode(int, int, int)
```

**Arguments:**

Type	Description
int	
int	
int	

---

**TransitNode Public Methods:**

```
AddSub(SubDomain)
```

**Arguments:**

Type	Description
SubDomain	

---

```
Node* GetTNode()
```

Return the transit node

**Return Value:**

Type	Description
Node*	Pointer to the transit node

---

```
SubList_t GetSubList()
```

Return the subs associated with this transit node

**Return Value:**

Type	Description
SubList_t	A vector of subdomains

---

```
SetSubList(SubList_t)
```

Set the Sub list to the passed list

**Arguments:**

Type	Description
SubList_t	A vector of subdomains

---

```
int GetMaxSubId()
```

This method is used to return the highest Sub id associated with this transit node

**Return Value:**

Type	Description
int	An integer

---

```
int GetId()
```

This method is used to return the transit node id

**Return Value:**

Type	Description
int	An integer

---

```
IPAddr_t GetIP()
```

This method is used to return the IP Address of the transit node

**Return Value:**

Type	Description
IPAddr_t	The IP address of the transit node

---

### 4.5.3 SubDomain

Class `SubDomain`. Defines an interface for the subdomain

**SubDomain Constructors:**

`SubDomain(double, int, Node)`

**Arguments:**

Type	Description
double	
int	
Node	

---

**SubDomain Public Methods:**

`AddNode(Node)`

**Arguments:**

Type	Description
Node	

---

`int GetSubId()`

This method is used to return the subdomain id

**Return Value:**

Type	Description
int	An integer (sub id)

---

`double GetSubTNodeid()`

This method is used to return the sub transit node id

**Return Value:**

Type	Description
double	A double (transit node id)

---

`NodeList_t GetNodeList()`

This method is used to return the list of the nodes associated with this subdomain

**Return Value:**

Type	Description
NodeList_t	A vector of nodes

---

`int GetMaxNodeId()`

This method is used to return the highest node id in the subdomain

**Return Value:**

Type	Description
int	An integer (highest node id)

---

### 4.5.4 Star Topology

Class `Star`. Define a star topology using point-to-point links

**Star Constructors:**

Star(Count\_t, IPAddr\_t, SystemId\_t)

Create a star topology with the specified number of leaf nodes. Uses the default point-to-point link object.

**Arguments:**

Type	Description
Count_t	Number of leaf nodes desired.
IPAddr_t	Starting <i>IP Address</i> of leaf nodes. <b>Default Value</b> is IPADDR_NONE
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

Star(Count\_t, const Linkp2p&, IPAddr\_t, SystemId\_t)

Create a star topology with the specified number of leaf nodes, and the specified point-to-point link object.

**Arguments:**

Type	Description
Count_t	Number of leaf nodes desired.
const Linkp2p&	Link object to use for leaf to core links.
IPAddr_t	Starting <i>IP Address</i> of leaf nodes. <b>Default Value</b> is IPADDR_NONE
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

Star(Count\_t, Node\*, IPAddr\_t, SystemId\_t)

Create a star topology with the specified number of leaf nodes. and an existing node for the core. Uses the default point-to-point link object.

**Arguments:**

Type	Description
Count_t	Number of leaf nodes desired.
Node*	Node pointer for core node.
IPAddr_t	Starting <i>IP Address</i> of leaf nodes. <b>Default Value</b> is IPADDR_NONE
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

Star(Count\_t, Node\*, const Linkp2p&, IPAddr\_t, SystemId\_t)

Create a star topology with the specified number of leaf nodes. and an existing node for the core. Uses the specified point-to-point link object for leaf links.

**Arguments:**

Type	Description
Count_t	Number of leaf nodes desired.
Node*	Node pointer for core node.
const Linkp2p&	Link object to use for leaf to core links.
IPAddr_t	Starting <i>IP Address</i> of leaf nodes. <b>Default Value</b> is IPADDR_NONE
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

**Star Public Methods:**

Node\* GetHub()

Get a pointer to the hub node in the star.

**Return Value:**

Type	Description
Node*	Node pointer to hub node.

---

Node\* GetLeaf(Count\_t)

Get a pointer to a leaf node.

**Return Value:**

Type	Description
Node*	Pointer to specified leaf node.

**Arguments:**

Type	Description
Count_t	Leaf node index to get.

---

Linkp2p\* HubLink(Count\_t)

Get a pointer to a hub to leaf link.

**Return Value:**

Type	Description
Linkp2p*	Pointer to point-to-point link from hub to specified leaf

**Arguments:**

Type	Description
Count_t	Leaf node index.

---

Linkp2p\* LeafLink(Count\_t)

Get a pointer to a leaf to hub link.

**Return Value:**

Type	Description
Linkp2p*	Pointer to point-to-point link from specified leaf to hub.

**Arguments:**

Type	Description
Count_t	Leaf node index.

---

Queue\* HubQueue(Count\_t)

Get a pointer to a hub to leaf queue.

**Return Value:**

Type	Description
Queue*	Pointer to queue from hub to specified leaf

**Arguments:**

Type	Description
Count_t	Leaf node index.

---

Queue\* LeafQueue(Count\_t)

Get a pointer to a leaf to hub queue.

**Return Value:**

Type	Description
Queue*	Pointer to queue from specified leaf to hub

**Arguments:**

Type	Description
Count_t	Leaf node index.

---

Count\_t LeafCount()

Get a count of the number of leaf nodes.

**Return Value:**

Type	Description
Count_t	Leaf node count.

---

BoundingBox(const Location&, const Location&, Angle\_t, Angle\_t, \*)

Define a bounding box for this star object. The bounding box is used to assign node locations, which are used during the simulation animation. If the hub node has an existing location, then this bounding box is relative to that node. Otherwise, the hub is placed in the center of the box, with all leaf nodes extending radially from the hub with length of half the bounding box size (minus a small fudge factor).

**Arguments:**

Type	Description
const Location&	Lower left corner of bounding box.
const Location&	Upper right corner of bounding box.
Angle_t	The angle that the radial links are centered around. <b>Default Value</b> is 0
Angle_t	The total size of the arc for the radial links <b>Default Value</b> is 2
*	

---

### 4.5.5 Grid Topology

Class `Grid`. Defines an object for a `m` by `n` grid of nodes. Grids are 2-dimensional array of nodes, with point-to-point links connecting nodes. The default p2p link is used. A starting IP Address may be specified, which is automatically incremented as new links are created. If `IPADDR_NONE` is specified, the ip addresses are not used.

#### Grid Constructors:

```
Grid(Count_t, Count_t, IPAddr_t, const Linkp2p&, SystemId_t)
```

Constructor for `Grid` object.

#### Arguments:

Type	Description
<code>Count_t</code>	Count of nodes on the grid x-axis.
<code>Count_t</code>	Count of nodes on the grid y-axis.
<code>IPAddr_t</code>	<i>IP Address</i> for node at <code>x = 0, y = 0</code> . <b>Default Value</b> is <code>IPADDR_NONE</code>
<code>const Linkp2p&amp;</code>	Link object to use for links. <b>Default Value</b> is <code>Linkp2p::Default</code>
<code>SystemId_t</code>	Responsible system ID for distributed simulation <b>Default Value</b> is <code>0</code>

---

#### Grid Public Methods:

```
Node* GetNode(Count_t, Count_t)
```

Get a pointer to a specified node in the grid.

#### Arguments:

Type	Description
<code>Count_t</code>	Node location X-coordinate. Node location Y-coordinate.
<code>Count_t</code>	

---

```
BoundingBox(const Location&, const Location&)
```

Define a bounding box for this grid object. The bounding box is used to assign node locations, which are used during the simulation animation.

#### Arguments:

Type	Description
<code>const Location&amp;</code>	Lower left corner of bounding box.
<code>const Location&amp;</code>	Upper right corner of bounding box.

---

An example of a simulation using the `Grid` topology object is given in program 4-1.

```

1 // Test program for Grid of nodes object
2 // George F. Riley. Georgia Tech, Spring 2002
3
4 #include <iostream>
5
6 #include "simulator.h"
7 #include "grid.h"
8 #include "rng.h"
9 #include "tcp-tahoe.h"
10 #include "application-tcpserver.h"
11 #include "application-tcpsend.h"
12
13 using namespace std;
14
15 int main(int argc, char** argv)
16 {
17     Count_t xs = 5;
18     Count_t ys = 5;
19     if (argc > 1) xs = atol(argv[1]);
20     if (argc > 2) ys = atol(argv[2]);
21     // Create the simulator object
22     Simulator s;
23
24     // Create the grid
25     Grid g(xs, ys, IPAddr("192.0.1.1"), 0, 0, 1, 1);
26     for (Count_t x = 0; x < xs; ++x)
27     {
28         Node* nb = g.GetNode(x,0); // Bottom row
29         Node* nt = g.GetNode(x,ys-1); // Top row
30         // Create a tcp server to use on top row
31         TCPServer* server = (TCPServer*)nt->AddApplication(TCPServer());
32         server->BindAndListen(nt, 80);
33         // Create a tcp application to connect to the servers
34         TCPSend* app = (TCPSend*)nb->AddApplication(
35             TCPSend(TCPTahoe(), nt->GetIPAddr(), 80, Constant(100000)));
36         app->Start(1.0 * x); // Staggered start
37     }
38     // Print some debug info for each
39     for (Count_t y = 0; y < ys; ++y)
40     {
41         for (Count_t x = 0; x < xs; ++x)
42         {
43             Node* n = g.GetNode(x, y);
44             cout << "Node " << x << " " << y
45                 << " location " << n->LocationX() << " " << n->LocationY()
46                 << " ip " << (string)IPAddr(n->GetIPAddr())
47                 << " number of ifaces " << n->InterfaceCount()
48                 << endl;
49         }
50     }
51     s.StopAt(10.0);
52     s.Progress(1.0);
53     s.Run(); // Run the simulation
54     cout << "Total events processed " << s.TotalEventsProcessed() << endl;
55     cout << "Setup time " << s.SetupTime() << endl;
56     cout << "Route time " << s.RouteTime() << endl;

```

Program 4-1 testgrid.cc

```

57     cout << "Run time "    << s.RunTime() << endl;
58 }
59
60

```

Program 4-1 examples/testgrid.cc (continued)

## 4.5.6 Dumbbell Topology

Class `Dumbbell`. Class `Dumbbell` creates a topology model with the familiar `dumbbell` configuration. Some number of nodes on the left and right side of the topology communicate through a single bottleneck link.

### Dumbbell Constructors:

`Dumbbell(Count_t, Count_t, Mult_t, SystemId_t)`

Constructor for `dumbbell`. The default point-to-point link (with the default bandwidth and delay) is used for the leaf node links. No *IP Address*s are assigned to any links.

#### Arguments:

Type	Description
<code>Count_t</code>	Count of nodes on the left side of the topology.
<code>Count_t</code>	Count of nodes on the right side of the topology.
<code>Mult_t</code>	Multiplier for the link bandwidth of the bottleneck link. For example, a multiplier of 0.1 indicates the bandwidth of the bottleneck is one tenth of the bandwidth of the leaf node links.
<code>SystemId_t</code>	Responsible system ID for distributed simulation <b>Default Value</b> is 0

`Dumbbell(Count_t, Count_t, Mult_t, const Linkp2p&, SystemId_t)`

Constructor for `dumbbell`, specifying a point-to-point link to use as a template for leaf node links. No *IP Address*s are assigned to any links.

#### Arguments:

Type	Description
<code>Count_t</code>	Count of nodes on the left side of the topology.
<code>Count_t</code>	Count of nodes on the right side of the topology.
<code>Mult_t</code>	Multiplier for the link bandwidth of the bottleneck link.
<code>const Linkp2p&amp;</code>	Link object to copy for leaf node links.
<code>SystemId_t</code>	Responsible system ID for distributed simulation <b>Default Value</b> is 0

`Dumbbell(Count_t, Count_t, Mult_t, IPAddr_t, IPAddr_t, SystemId_t)`

Constructor for `dumbbell`. The default point-to-point link (with the default bandwidth and delay) is used for the leaf node links. *IP Address*s are assigned to leaf nodes based on the arguments specified.

#### Arguments:

Type	Description
<code>Count_t</code>	Count of nodes on the left side of the topology.
<code>Count_t</code>	Count of nodes on the right side of the topology.
<code>Mult_t</code>	Multiplier for the link bandwidth of the bottleneck link.
<code>IPAddr_t</code>	Starting <i>IP Address</i> for left side leaf nodes. The <i>IP Address</i> is incremented by one for each created left side leaf.
<code>IPAddr_t</code>	Starting <i>IP Address</i> for right side leaf nodes. The <i>IP Address</i> is incremented by one for each created right side leaf.
<code>SystemId_t</code>	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

Dumbbell(Count\_t, Count\_t, Mult\_t, IPAddr\_t, IPAddr\_t, const Linkp2p&, SystemId\_t)  
 Constructor for dumbbell, specifying a point-to-point link to use as a template for leaf node links, and *IP Address*s for leaf nodes.

**Arguments:**

Type	Description
Count_t	Count of nodes on the left side of the topology.
Count_t	Count of nodes on the right side of the topology.
Mult_t	Multiplier for the link bandwidth of the bottleneck link.
IPAddr_t	Starting <i>IP Address</i> for left side leaf nodes. The <i>IP Address</i> is incremented by one for each created left side leaf.
IPAddr_t	Starting <i>IP Address</i> for right side leaf nodes. The <i>IP Address</i> is incremented by one for each created right side leaf.
const Linkp2p&	Link object to copy for leaf node links.
SystemId_t	Responsible system ID for distributed simulation <b>Default Value</b> is 0

---

**Dumbbell Public Methods:**

Node\* Left(Count\_t)  
 Get a pointer to a left side leaf node.

**Return Value:**

Type	Description
Node*	Pointer to the specified left side leaf node.

**Arguments:**

Type	Description
Count_t	Index for left side node to return.

---

Node\* Right(Count\_t)  
 Get a pointer to a right side leaf node.

**Return Value:**

Type	Description
Node*	Pointer to the specified right side leaf node.

**Arguments:**

Type	Description
Count_t	Index for right side node to return.

---

Node\* Left()  
 Get a pointer to the left side bottleneck router.

**Return Value:**

Type	Description
Node*	Pointer to left side bottleneck router.

---

Node\* Right()  
 Get a pointer to the right side bottleneck router.

**Return Value:**

Type	Description
Node*	Pointer to right side bottleneck router.

---

Linkp2p\* LeftLink(Count\_t)

Get a pointer to the link from the specified leaf node to bottleneck router.

**Return Value:**

Type	Description
Linkp2p*	Pointer to specified left side leaf to router link.

**Arguments:**

Type	Description
Count_t	Index for left side node.

---

Linkp2p\* RightLink(Count\_t)

Get a pointer to the link from the specified left side leaf node to bottleneck router.

**Return Value:**

Type	Description
Linkp2p*	Pointer to specified right side leaf to router link.

**Arguments:**

Type	Description
Count_t	Index for right side node.

---

Linkp2p\* LeftLink()

Get a pointer to the link from the left side router to right side router.

**Return Value:**

Type	Description
Linkp2p*	Pointer to specified router to router link.

---

Linkp2p\* RightLink()

Get a pointer to the link from the right side router to left side router.

**Return Value:**

Type	Description
Linkp2p*	Pointer to specified router to router link.

---

Queue\* LeftQueue(Count\_t)

Get a pointer to the queue from the specified left leaf node to bottleneck router.

**Return Value:**

Type	Description
Queue*	Pointer to specified left side leaf to router queue.

**Arguments:**

Type	Description
Count_t	Index for left side node.

---

Queue\* RightQueue(Count\_t)

Get a pointer to the queue from the specified right leaf node to bottleneck router.

**Return Value:**

Type	Description
Queue*	Pointer to specified left side leaf to router queue.

**Arguments:**

Type	Description
Count_t	Index for right side node.

---

Queue\* LeftQueue()

Get a pointer to the queue from the left side router to right side router.

**Return Value:**

Type	Description
Queue*	Pointer to specified router to router queue.

---

```
Queue* RightQueue()
```

Get a pointer to the queue from the right side router to left side router.

**Return Value:**

Type	Description
Queue*	Pointer to specified router to router queue.

---

```
Count_t LeftCount()
```

Return the count of the number left side leaf nodes. //Doc:Arg1 Count of left side leaf nodes.

---

```
Count_t RightCount()
```

Return the count of the number right side leaf nodes. //Doc:Arg1 Count of right side leaf nodes.

---

```
BoundingBox(const Location&, const Location&)
```

Define a bounding box for this dumbbell object. The bounding box is used to assign node locations, which are used during the simulation animation.

**Arguments:**

Type	Description
const Location&	Lower left corner of bounding box.
const Location&	Upper right corner of bounding box.

---

An example of a simulation using the Dumbbell topology object is given in program 4-2.

```

1 // Test the dumbbell topology stock object.
2 // George F. Riley. Georgia Tech, Fall 2002
3
4 #include "simulator.h"
5 #include "dumbbell.h"
6 #include "linkp2p.h"
7 #include "ratetimestat.h"
8 #include "tcp-tahoe.h"
9 #include "application-tcpsend.h"
10 #include "application-tcpserver.h"
11 #include "globalstats.h"
12 #include "droptail.h"
13 #include "ipv4.h"
14
15 using namespace std;
16
17
18 int main()
19 {
20     Simulator s;
21
22     Queue::Default(DropTail()); // Set default to non-detailed drop tail
23     Trace* gs = Trace::Instance();
24     gs->IPDotted(true);
25     gs->Open("testbell.txt");
26     TCP::LogFlagsText(true); // Log flags in text mode
27     // Enable all 13 tracing
28     IPV4::Instance()->SetTrace(Trace::ENABLED);
29
30     Linkp2p l;
31     l.Bandwidth(Rate("100Mb"));
32     Dumbbell b(10, 10, 0.1, IPAddr("192.168.1.0"), IPAddr("192.168.2.0"), 1);
33     for (Count_t i = 0; i < b.LeftCount(); ++i)
34     {
35         cout << "Left node " << i
36              << " ipaddr " << (string)IPAddr(b.Left(i)->GetIPAddr()) << endl;
37     }
38     for (Count_t i = 0; i < b.RightCount(); ++i)
39     {
40         cout << "Right node " << i
41              << " ipaddr " << (string)IPAddr(b.Right(i)->GetIPAddr()) << endl;
42         // Add a tcp server on right side
43         Node* n = b.Right(i);
44         TCPServer* server = (TCPServer*)n->AddApplication(TCPServer());
45         server->BindAndListen(n, 40);
46     }
47     TCPSend* app = (TCPSend*)b.Left(0)->AddApplication(
48         TCPSend(TCPTahoe(), b.Right(0)->GetIPAddr(), 40, Constant(100000)));
49     app->l4Proto->SetTrace(Trace::ENABLED);
50     app->Start(0.1);
51     s.Run();
52     Stats::Print(); // Print the statistics
53     cout << "Memory usage after run "
54          << s.ReportMemoryUsageMB() << "MB" << endl;
55     cout << "Total events processed " << s.TotalEventsProcessed() << endl;
56     cout << "Setup time " << s.SetupTime() << endl;

```

Program 4-2 testbell.cc

```

57     cout << "Route time " << s.RouteTime() << endl;
58     cout << "Run time " << s.RunTime() << endl;
59     gs->Close();
60 }

```

Program 4-2 examples/testbell.cc (continued)

## 4.5.7 Tree Topology

Class `Tree`. Defines a tree topology using point-to-point links.

### Tree Constructors:

`Tree(Count_t, Count_t, IPAddr_t, Mask_t, SystemId_t)`

Construct a `Tree` topology object with the specified number of levels, fanout, *IP Address* and address mask. The default point-to-point link object is used for links.

#### Arguments:

Type	Description
<code>Count_t</code>	Number of levels in the tree.
<code>Count_t</code>	Fanout of each level (number of child nodes for each parent).
<code>IPAddr_t</code>	Starting <i>IP Address</i> . The <i>IP Address</i> is incremented by one for each node in the tree. <b>Default Value</b> is <code>IPADDR_NONE</code>
<code>Mask_t</code>	<i>IP Address</i> mask. <b>Default Value</b> is <code>Mask32</code>
<code>SystemId_t</code>	Responsible system ID for distributed simulation <b>Default Value</b> is 0

`Tree(Count_t, Count_t, const Linkp2p&, IPAddr_t, Mask_t, SystemId_t)`

Construct a `Tree` topology object with the specified number of levels, fanout, *IP Address* and address mask, and the specified point-to-point link.

#### Arguments:

Type	Description
<code>Count_t</code>	Number of levels in the tree.
<code>Count_t</code>	Fanout of each level (number of child nodes for each parent).
<code>const Linkp2p&amp;</code>	A reference to a point-to-point link object to copy for each link in the tree.
<code>IPAddr_t</code>	Starting <i>IP Address</i> . The <i>IP Address</i> is incremented by one for each <b>Default Value</b> is <code>IPADDR_NONE</code>
<code>Mask_t</code>	node in the tree. <b>Default Value</b> is <code>Mask32</code>
<code>SystemId_t</code>	<i>IP Address</i> mask. <b>Default Value</b> is 0

### Tree Public Methods:

`Node* GetRoot()`

Return a pointer to the root node of the tree.

#### Return Value:

Type	Description
<code>Node*</code>	Pointer to root node.

`Node* GetNode(Count_t, Count_t)`

Return a pointer to the node at the specified level, index.

#### Return Value:

Type	Description
<code>Node*</code>	Pointer to the specified node.

#### Arguments:

Type	Description
<code>Count_t</code>	Level number (root is level 0).
<code>Count_t</code>	Index, range 0 to number of nodes in the level - 1.

---

Node\* GetLeaf(Count\_t)

Return a pointer to the specified leaf node.

**Return Value:**

Type	Description
Node*	Pointer to the specified leaf node.

**Arguments:**

Type	Description
Count_t	Desired leaf node indes, counting from zero.

---

Linkp2p\* GetChildLink(Count\_t, Count\_t, Count\_t)

Get a pointer to the link from the specified node to its specified child.

**Return Value:**

Type	Description
Linkp2p*	Pointer to the specified link.

**Arguments:**

Type	Description
Count_t	Level number (root is level 0).
Count_t	Index, range 0 to number of nodes in the level - 1.
Count_t	Child index, range 0 to fanout - 1.

---

Linkp2p\* GetParentLink(Count\_t, Count\_t)

Get a pointer to the link from the specified node to its parent.

**Return Value:**

Type	Description
Linkp2p*	Pointer to the specified link.

**Arguments:**

Type	Description
Count_t	Level number (root is level 0).
Count_t	Index, range 0 to number of nodes in the level - 1.

---

Queue\* GetChildQueue(Count\_t, Count\_t, Count\_t)

Get a pointer to the queue from the specified node to its specified child.

**Return Value:**

Type	Description
Queue*	Pointer to the specified queue.

**Arguments:**

Type	Description
Count_t	Level number (root is level 0).
Count_t	Index, range 0 to number of nodes in the level - 1.
Count_t	Child index, range 0 to fanout - 1.

---

Queue\* GetParentQueue(Count\_t, Count\_t)

Get a pointer to the queue from the specified node to its parent.

**Return Value:**

Type	Description
Queue*	Pointer to the specified queue.

**Arguments:**

Type	Description
Count_t	Level number (root is level 0).
Count_t	Index, range 0 to number of nodes in the level - 1.

---

```
Count_t LeafCount()
```

Get a count of the number of leaf nodes in this tree.

**Return Value:**

Type	Description
Count_t	Count of leaf nodes.

---

```
BoundingBox(const Location&, const Location&)
```

Define a bounding box for this tree object. The bounding box is used to assign node locations, which are used during the simulation animation.

**Arguments:**

Type	Description
const Location&	Lower left corner of bounding box.
const Location&	Upper right corner of bounding box.

---

### 4.5.8 Random Topology

*To be supplied*

## Chapter 5

# Defining Applications

Once a topology is defined, the simulation must then describe the flow of data through the defined network topology. In *GTNets*, this is done by creating *Applications* at various nodes, which in turn generate data demands on the network specific to the application behavior. *GTNets* defines a number of application models, described below.

### 5.1 Application Base Class

**Class Application.** Class `Application` is the base class for all *GTNets* applications. It defines the interface between the application class and the associated layer 4 protocols. Applications can have one or more layer 4 protocols assigned, to allow (for example) a web browser model with multiple simultaneous connections.

#### Application Constructors:

```
Application()
```

The default constructor for the `Application` class.

---

```
Application(const Application&)
```

Copy constructor, used by the `Copy()` method for all applications.

#### Arguments:

Type	Description
<code>const Application&amp;</code>	Application object to copy.

---

#### Application Public Methods:

```
Start(Time_t)
```

Schedule a `Start` event for this application at the specified time.

#### Arguments:

Type	Description
<code>Time_t</code>	The simulation time to start the application. The <code>StartApp</code> method of the application will be called at the specified time.

---

```
Stop(Time_t)
```

Schedule a `Stop` event for this application at the specified time.

#### Arguments:

Type	Description
<code>Time_t</code>	The simulation time to stop the application. The <code>StopApp</code> method of the application will be called at the specified time.

---

Receive(Packet\*, L4Protocol\*, Seq\_t)

Called by an associated layer 4 protocol when data is received.

**Arguments:**

Type	Description
Packet*	The packet received, with the layer 4 PDU removed.
L4Protocol*	A pointer to the layer 4 protocol object that recieved the data.
Seq_t	Optional sequence number for this data (TCP protocols only) <b>Default Value</b> is 0

---

Sent(Count\_t, L4Protocol\*)

Called by an associated layer 4 protocol when all some part of the outstanding data has been sent. For TCP protocols, this occurs when the acknowledgement is received from the peer.

**Arguments:**

Type	Description
Count_t	Count of number of bytes successfully sent.
L4Protocol*	A pointer to the layer 4 protocol object that sent the data.

---

CloseRequest(L4Protocol\*)

Called by an associated layer 4 protocol when a connection close request has been received from a peer. Applications should respond by calling the corresponding layer 4 ::Close() routine.

**Arguments:**

Type	Description
L4Protocol*	A pointer to the layer 4 protocol object that closed.

---

Closed(L4Protocol\*)

Called by an associated layer 4 protocol when a connection has completely closed.

**Arguments:**

Type	Description
L4Protocol*	A pointer to the layer 4 protocol object that closed.

---

ConnectionComplete(L4Protocol\*)

Called by an associated layer 4 protocol when a previous connection request has successfully completed.

**Arguments:**

Type	Description
L4Protocol*	A pointer to the layer 4 protocol object that connected.

---

ConnectionFailed(L4Protocol\*)

Called by an associated layer 4 protocol when a previous connection request has failed.

**Arguments:**

Type	Description
L4Protocol*	A pointer to the layer 4 protocol object that failed.

---

bool ConnectionFromPeer(L4Protocol\*, IPAddr\_t, PortId\_t)

Called by an associated layer 4 protocol when a listening TCP protocol has received a connection request.

**Return Value:**

Type	Description
bool	Return true if able to accept a new connection, false if not.

**Arguments:**

Type	Description
L4Protocol*	A pointer to the listening TCP protocol object.
IPAddr_t	<i>IP Address</i> of remote peer.
PortId_t	Port number of report peer.

---

DeleteOnComplete(bool)

Specifies that this application object should automatically be deleted when the application has finished.

**Arguments:**

Type	Description
------	-------------

bool	True if delete on complete desired. <b>Default Value</b> is true
------	--

---

CopyOnConnect(bool)

Specifies that this application object should be copied when connection requests are processed.

**Arguments:**

Type	Description
------	-------------

bool	True if delete on copy on connect desired.
------	--

---

bool CopyOnConnect()

Return current status of copy on connect flag.

**Return Value:**

Type	Description
------	-------------

bool	True if copy on connect, false if not.
------	--

---

StartApp()

Called at the specified application start time.

---

StopApp()

Called at the specified application stop time.

---

AttachNode(Node\*)

Specify which node to which this application is attached.

**Arguments:**

Type	Description
------	-------------

Node*	Node pointer to attached node.
-------	--------------------------------

---

Application\* Copy()

Return a copy of this application. This method must be defined by all subclasses.

**Return Value:**

Type	Description
------	-------------

Application*	A pointer to a new copy of this application.
--------------	--

---

L4Protocol\* GetL4()

Returns a pointer to the layer 4 protocol object, if this application has a single associated l4 protocol.

**Return Value:**

Type	Description
------	-------------

L4Protocol*	Pointer to the single layer 4 object, or nil if none, or nil if more than one.
-------------	--

---

Application **Public Members:**

Type	Name	Description
bool	deleteOnComplete	True if delete on complete requested.
bool	copyOnConnect	True if copy on connect requested.

---

## 5.2 CBRAApplication

Class CBRAApplication. Define a application that generates constant bit rate data between two nodes.

### CBRAApplication Constructors:

CBRAApplication(Node\*, Node\*)

Specifies the two node endpoints. Uses UDP for the layer 4 protocol, and default 500,000 bits/sec for the rate and 512 for the packet size.

#### Arguments:

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint

---

CBRAApplication(Node\*, Node\*, const L4Protocol&)

Specifies the two node endpoints. and a layer 4 protocol object to copy. Uses default 500,000 bits/sec for the rate and 512 for the packet size.

#### Arguments:

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
const L4Protocol&	A layer 4 protocol object to copy for the protocol endpoints.

---

CBRAApplication(Node\*, Node\*, const L4Protocol&, Rate\_t)

Specifies the two node endpoints, a layer 4 protocol object to copy, and the constant bit rate. Uses 512 for the packet size.

#### Arguments:

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
const L4Protocol&	A layer 4 protocol object to copy for the protocol endpoints.
Rate_t	Rate to use for the constant bit rate data generator.

---

CBRAApplication(Node\*, Node\*, const L4Protocol&, Rate\_t, Size\_t)

Specifies the two node endpoints, a layer 4 protocol object to copy, the constant bit rate, and the packet size.

#### Arguments:

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
const L4Protocol&	A layer 4 protocol object to copy for the protocol endpoints.
Rate_t	Rate to use for the constant bit rate data generator.
Size_t	Packet size to use.

---

CBRAApplication(Node\*, Node\*, Rate\_t)

Specifies the two node endpoints and the constant bit rate. Uses a UDP object for the layer 4 protocol.

#### Arguments:

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
Rate_t	Rate to use for the constant bit rate data generator.

---

```
CBRAApplication(Node*, Node*, Rate_t, Size_t)
```

Specifies the two node endpoints, the constant bit rate, and the packet size. Uses a UDP object for the layer 4 protocol.

**Arguments:**

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
Rate_t	Rate to use for the constant bit rate data generator.
Size_t	Packet size to use.

---

### 5.3 OnOffApplication

Class OnOffApplication. Defines an application that uses the on-off data generator model.

OnOffApplication **Constructors:**

```
OnOffApplication(Node*, Node*, const Random&, const L4Protocol&, Rate_t, Size_t)
```

Constructor specifying both endpoint nodes, a single random number generator for both the *On* and *Off* distributions, and a layer 4 protocol object. Optionally specifies bit rate (when *On* (defaults to a globally specified default rate), and optionally a packet size (defaults to a globally specified default size). See `SetDefaultRate` and `SetDefaultSize` below.

**Arguments:**

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
const Random&	Random number generator to use for both <i>On</i> and <i>Off</i> time periods.
const L4Protocol&	A layer 4 protocol object to copy.
Rate_t	Data rate to generate when <i>On</i> . <b>Default Value</b> is <code>defaultRate</code>
Size_t	Packet size. <b>Default Value</b> is <code>defaultSize</code>

---

```
OnOffApplication(Node*, Node*, const Random&, const Random&, const L4Protocol&, Rate_t, Size_t)
```

Constructor specifying both endpoint nodes, a random number generator for the *On* and a second random number generator for the *Off* distribution, and a layer 4 protocol object. Optionally specifies bit rate (when *On* (defaults to a globally specified default rate), and optionally a packet size (defaults to a globally specified default size). See `SetDefaultRate` and `SetDefaultSize` below.

**Arguments:**

Type	Description
Node*	Pointer to first node endpoint
Node*	Pointer to second node endpoint
const Random&	Random number generator to use for <i>On</i> period.
const Random&	Random number generator to use for <i>Off</i> period.
const L4Protocol&	A layer 4 protocol object to copy.
Rate_t	Data rate to generate when <i>On</i> . <b>Default Value</b> is <code>defaultRate</code>
Size_t	Packet size. <b>Default Value</b> is <code>defaultSize</code>

---

**OnOffApplication Public Methods:**

MaxBytes(Count\_t)

Specify a maximum number of bytes to send, after which the application shuts down.

**Arguments:**

Type	Description
Count_t	Maximum number of bytes to send.

---

**OnOffApplication Static Methods:**

DefaultRate(Rate\_t)

Specify a default data rate to use for all On/Off applications by default

**Arguments:**

Type	Description
Rate_t	The default data rate (when the application is <i>On</i> ).

---

DefaultSize(Size\_t)

Specifies a default packet size to use for all On/Off applications by default.

**Arguments:**

Type	Description
Size_t	The default packet size.

---

## 5.4 TCPApplication

Class TCPApplication. Class TCPApplication defines a virtual base class that can be subclassed by any application that has a single TCP connection endpoint.

**TCPApplication Constructors:**

TCPApplication(const TCP&)

**Arguments:**

Type	Description
const TCP&	Reference to any TCP variation to use for the TCP connection.

---

TCPApplication(const TCPApplication&)

Copy Constructor.

**Arguments:**

Type	Description
const TCPApplication&	TCPApplication to copy.

---

**TCPApplication Public Methods:**

AttachNode(Node\*)

Specify which node to which this application is attached.

**Arguments:**

Type	Description
Node*	Node pointer to attached node.

---

L4Protocol\* GetL4()

Get a pointer to the single TCP object for this application.

**Return Value:**

Type	Description
L4Protocol*	Pointer to associated TCP object.

---

Bind(PortId\_t)

Bind the TCP object to a specified port.

**Arguments:**

Type	Description
PortId_t	Port number to bind. Specify zero if an available port is to be chosen.

---

CloseOnEmpty(bool)

Specify that the TCP object should close the connection automatically when all pending data has been sent.

**Arguments:**

Type	Description
bool	True of close on empty desired. <b>Default Value</b> is true

---

SetTrace(Trace::TraceStatus)

Set the trace status of the associated TCP endpoint.

**Arguments:**

Type	Description
Trace::TraceStatus	Trace status desired.

---

## 5.5 TCPReceive

Class TCPReceive. Class TCPReceive defines an application that sends a request to a TCPServer object, and awaits a reply from the server.

TCPReceive **Constructors:**

TCPReceive(IPAddr\_t, PortId\_t, const Random&, const Random&, const TCP&, const Random&, Count\_t)

Constructor for TCPReceive objects.

**Arguments:**

Type	Description
IPAddr_t	<i>IP Address</i> of the TCPServer to connect to.
PortId_t	Port number of the of the TCPServer to connect to.
const Random&	Random variable specifying the size of the request data for each iteration Use a Constant random variable if a fixed size is desired.
const Random&	Random variable specifying the size of the server reply data for each iteration Use a Constant random variable if a fixed size is desired.
const TCP&	A reference to any TCP variant to use for the TCP endpoint. <b>Default Value</b> is TCP::Default
const Random&	Random variable specifying the sleep time between sending iterations. <b>Default Value</b> is Constant0
Count_t	Limit on the number of times the data is sent. <b>Default Value</b> is 1

---

TCPReceive(const TCPReceive&)

Copy Constructor.

**Arguments:**

Type	Description
const TCPReceive&	TCPReceive object to copy.

---

## 5.6 TCPServer

Class `TCPServer`. Class `TCPServer` creates a simple model of a request/response TCP server. The server binds to a specified port, and listens for connection requests from TCP peers. The data received from the peers specifies how much data to send or receive.

### TCPServer Constructors:

`TCPServer(const TCP&)`

Constructor for `TCPServer` objects. Optionally specify a reference to any TCP variant to use for the listening TCP endpoint.

#### Arguments:

Type	Description
<code>const TCP&amp;</code>	A reference to any TCP variant to use for the listening endpoint. <b>Default Value</b> is <code>TCP::Default</code>

`TCPServer(const TCPServer&)`

Copy constructor.

#### Arguments:

Type	Description
<code>const TCPServer&amp;</code>	<code>TCPServer</code> object to copy.

### TCPServer Public Methods:

`BindAndListen(Node*, PortId_t)`

Bind the application to the specified port and start listening for connections.

#### Arguments:

Type	Description
<code>Node*</code>	Node pointer to the node this application is associated with.
<code>PortId_t</code>	Port number to bind to.

`BindAndListen(PortId_t)`

Bind the application to the specified port and start listening for connections. Node is already known.

#### Arguments:

Type	Description
<code>PortId_t</code>	Port number to bind to.

`SetStatistics(Statistics*)`

Set a statistics object to use for statistics collection. This application will record the response time of each message sent to the statistics object.

#### Arguments:

Type	Description
<code>Statistics*</code>	Pointer to a statistics object.

### TCPServer Public Members:

Type	Name	Description
<code>Count_t</code>	<code>totSent</code>	Count of the total number of bytes sent by this application.
<code>Count_t</code>	<code>totAck</code>	Count of total bytes acknowledged by this application.
<code>Count_t</code>	<code>totRx</code>	Count of total bytes received by this application.
<code>Time_t</code>	<code>msgStart</code>	Time the last message started transmission or receipt.
<code>bool</code>	<code>timeKnown</code>	True if the start time of the message is known.

## 5.7 WebServer

Class `WebServer`. Class `WebServer` defines the behavior of a web server application, including the Gnutella `GCache` scripts.

### WebServer Constructors:

`WebServer(const TCP&)`

Constructor for `WebServer` object. Single (optional) argument is a reference to a `TCP` object to use for the listening `TCP` endpoint.

#### Arguments:

Type	Description
<code>const TCP&amp;</code>	Reference to a <code>TCP</code> object to copy for listening endpoint. <b>Default Value</b> is <code>TCP::Default</code>

`WebServer(const WebServer&)`

Copy constructor

#### Arguments:

Type	Description
<code>const WebServer&amp;</code>	Object to copy

### WebServer Public Methods:

`EnableGCache(bool)`

Specify enabling or disabling of the Gnutella `GCache` processing at this web server.

#### Arguments:

Type	Description
<code>bool</code>	True if <code>GCache</code> processing enabled. <b>Default Value</b> is <code>true</code>

`EnableLog(bool)`

Specify enabling or disabling of the connection count versus time history logging.

#### Arguments:

Type	Description
<code>bool</code>	True if count vs. time logging desired. <b>Default Value</b> is <code>true</code>

`MaxConnections(Count_t)`

Specify a limit on the maximum number of simultaneous connections that can be open for this server.

#### Arguments:

Type	Description
<code>Count_t</code>	Connection limit.

`PrintTimeCount(std::ostream&, char, ')`

Print the count versus time history log on an output stream.

#### Arguments:

Type	Description
<code>std::ostream&amp;</code>	Output stream on which to log the count versus time history.
<code>char</code>	Separator character to use. <b>Default Value</b> is <code>'</code>

`MaxIP(Count_t)`

Specify the maximum number of *IP Address*s to log in the `GCache`. If more than this limit are posted, the earlier ones roll off.

#### Arguments:

Type	Description
<code>Count_t</code>	Limit on the number of posted <i>IP Address</i> s

---

MaxURL(Count\_t)

Specify the maximum number of URLs to log in the GCACHE. If more than this limit are posted, the earlier ones roll off.

**Arguments:**

Type	Description
Count_t	Limit on the number of posted URLs

---

WebServer **Static Methods:**

Port(PortId\_t)

Specify a non-default port to use for newly created WebServer objects.

**Arguments:**

Type	Description
PortId_t	Port number.

---

DefaultMaxConnections(Count\_t)

Specify a default limit on the maximum number of simultaneous connections for newly created WebServer objects.

**Arguments:**

Type	Description
Count_t	Connection limit.

---

DefaultMaxIP(Count\_t)

Specify a default limit on the number of GCACHE posted *IP Address*s for newly created WebServer objects.

**Arguments:**

Type	Description
Count_t	GCACHE IPA limit.

---

DefaultMaxURL(Count\_t)

Specify a default limit on the number of GCACHE posted URLs for newly created WebServer objects.

**Arguments:**

Type	Description
Count_t	GCACHE URL limit.

---

## 5.8 WebBrowser

Class `WebBrowser`. Simulation model of a web browser. The browser behavior uses empirical distributions (based on a study by Bruce Mah published in Infocom 1997) to define web browsing actions. Each browser performs the following actions. First a server is selected randomly from the server list passed in the constructor, and a connection with that server is established. Then the number of objects (individual connections) needed to get the web page is sampled from the empirical distributions. For each object needed, a request size and a response size are sampled from the empirical distribution, and the request is sent to the server. If more than one object is needed, a second simultaneous connection is opened and a second request is sent. When the entire request has been received by a web server, the number of bytes from the response size are sent to the browser, and the connection is closed. When the browser receives all the bytes of the requested object and detects the connection closing, another object is requested (if any remain for this page). After all of the objects for a given page are received, the browser "sleeps" for a period of time (sampled from the empirical distribution). After the sleep time is over, another web page (again with multiple objects) is requested, either from the same server or a new one, depending on the empirical random distribution of *Consecutive Pages*, which indicates how many times the same server is accessed consecutively.

**WebBrowser Constructors:**

```
WebBrowser(const IPAddrVec_t&, const Random&, const TCP&)
```

Constructor for Web Browser objects.

**Arguments:**

Type	Description
const IPAddrVec_t&	A vector of <i>IP Address</i> s to sample for selecting a web server.
const Random&	A random number generator to sample for selecting a web server.
const TCP&	A reference to a TCP variant to use for all connections. <b>Default Value</b> is TCP::Default

```
WebBrowser(const WebBrowser&)
```

Copy Constructor

**Arguments:**

Type	Description
const WebBrowser&	Web Browser object to copy.

**WebBrowser Public Methods:**

```
TCP* GetTCP()
```

Get the TCP prototype used for this browser.

**Return Value:**

Type	Description
TCP*	Pointer to TCP prototype

```
ConcurrentConnections(Count_t)
```

Specifies the number of concurrent connections for this browser.

**Arguments:**

Type	Description
Count_t	Count of concurrent connections.

```
SetStatistics(Statistics*)
```

Set a statistics object used for logging web response time.

**Arguments:**

Type	Description
Statistics*	A pointer to any statistics object (such as a histogram).

```
IdleTime(Time_t)
```

Specify the idle time. The browser will stop initiating requests at this time, allowing the simulation to (eventually) finish.

**Arguments:**

Type	Description
Time_t	Idle time in simulation seconds.

```
ThinkTimeBound(Time_t)
```

Specify a bound on the think time. The empirical distributions for think time allow a fairly large think time, which can result in a simulation where most of the browsers are thinking. This allows an arbitrary upper bound on the think time.

**Arguments:**

Type	Description
Time_t	Think time bound in simulation seconds.

**WebBrowser Static Methods:**`InitializeRandom(Seed_t)`

Specify a seed for the random number generators.

**Arguments:**

<b>Type</b>	<b>Description</b>
<code>Seed_t</code>	Random seed. <b>Default Value</b> is <code>Random::RANDOM_SEED</code>

---

# Chapter 6

## Layer 4 Protocols

*GTNets* provides simulation models of both the *TCP* and *UDP* protocols for layer 4 processing. For *TCP*, the simulator has three different variations:

1. TCP Tahoe
2. TCP Reno
3. TCP New Reno

The base class `L4Protocol` defines the basic functionality required by all layer 4 protocols. The base class `TCP` defines the *TCP* functionality, and in fact nearly all of the implementation for the three variants is found in the base class. Further, the *TCP* implementations in *GTNets* allow for the specification any round-trip time estimator.

The classes defining the layer 4 protocols and the round-trip time estimators are defined below.

### 6.1 L4Protocol Base Class

Class `L4Protocol`. Class `L4Protocol` is a virtual base class defines the interface for all of the layer 4 protocols in *GTNets*.

#### `L4Protocol` Constructors:

`L4Protocol()`  
Default constructor, no arguments.

---

`L4Protocol(Node*)`  
Construct a layer 4 protocol and assign it to the specified node.

#### Arguments:

Type	Description
<code>Node*</code>	Pointer to node to which to assign this protocol.

---

`L4Protocol(const L4Protocol&)`  
Copy constructor.

#### Arguments:

Type	Description
<code>const L4Protocol&amp;</code>	<code>L4Protocol</code> object to copy.

---

L4Protocol **Public Methods:**

Handle(Event\*, Time\_t)

The layer 4 protocol class is a subclass of `Handler`, since it must handle events, specifically the delayed transmit and receive events for processing "ExtraDelay".

**Arguments:**

Type	Description
Event*	A pointer to the event being handled.
Time_t	The current simulation time.

---

Layer\_t Layer()

All protocol objects in *GTNets* must define the `Layer` method, returning the protocol layer for this protocol.

**Return Value:**

Type	Description
Layer_t	Protocol layer number for this L4 protocol (always 4).

---

DataIndication(Node\*, Packet\*, IPAddr\_t, Interface\*)

Indicates that a packet has been received by the layer 3 protocol addressed to this node, and the protocol number in the L3 header matches the registered protocol number for this protocol.

**Arguments:**

Type	Description
Node*	Node pointer to associated node.
Packet*	Pointer to the received packet. The L3 PDU has been removed,
IPAddr_t	so the L4 PDU is the top of the PDU stack. <i>IP Address</i> of destination, since nodes can have multiple <i>IP Address</i> s.
Interface*	Interface from which this packet was received.

---

NCount\_t Send(Size\_t)

Send the specified amount of data. This version of `Send` does not actually use data contents, just an indication of the amount of data. This is useful for applications where data contents is not needed for the simulation.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
Size_t	Number of bytes to send.

---

NCount\_t Send(char\*, Size\_t)

Send the specified data. This version uses both data size and data contents.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
char*	Pointer to data to send.
Size_t	Number of bytes to send.

---

```
NCount_t Send(Data&)
```

Send the specified Data PDU.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
Data&	Reference to the Data PDU to send. Data PDUs have size and an optional contents field.

---

```
NCount_t SendTo(Size_t, IPAddr_t, PortId_t)
```

Send the specified number of bytes to the specified *IP Address* and port. Does not include data contents.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
Size_t	Number of bytes to send.
IPAddr_t	Destination <i>IP Address</i> .
PortId_t	Destination port.

---

```
NCount_t SendTo(char*, Size_t, IPAddr_t, PortId_t)
```

Send the specified number of bytes with associated data contents to the specified *IP Address* and port.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
char*	Number of bytes to send.
Size_t	Pointer to data to send.
IPAddr_t	Destination <i>IP Address</i> .
PortId_t	Destination port.

---

```
NCount_t SendTo(Data&, IPAddr_t, PortId_t)
```

Send a Data PDU to the specified destination.

**Return Value:**

Type	Description
NCount_t	Number of bytes actually sent.

**Arguments:**

Type	Description
Data&	Reference to the Data PDU to send. Data PDUs have size and an optional contents field.
IPAddr_t	Destination <i>IP Address</i> .
PortId_t	Destination port.

---

```
bool Bind(PortId_t)
```

Bind this L4 protocol to the specified port on the the attached node.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
PortId_t	Port number to bind.

---

```
bool Bind()
```

Bind this L4 protocol to any available port on the attached node.

**Return Value:**

Type	Description
bool	True if successful.

---

```
bool Unbind(Proto_t, PortId_t)
```

Remove a port binding.

**Return Value:**

Type	Description
bool	True if successful.

**Arguments:**

Type	Description
Proto_t	Protocol number for this L4 object.
PortId_t	Port number to remove binding for.

---

```
Attach(Node*)
```

Attach this L4 object to the specified node.

**Arguments:**

Type	Description
Node*	Pointer to node to attach.

---

```
PortId_t Port()
```

Return the currently bound port number.

**Return Value:**

Type	Description
PortId_t	Currently bound port, or PORT_NONE if not bound.

---

```
Proto_t Proto()
```

Return the protocol number for this layer 4 protocol

**Return Value:**

Type	Description
Proto_t	Protocol number

---

```
bool Connect(IPAddr_t, PortId_t)
```

Initiate a connection to a remote host.

**Return Value:**

Type	Description
bool	True if connection initiated ok. This does not mean the connection was successful if a TCP L4 object.

**Arguments:**

Type	Description
IPAddr_t	IP Address of remote host.
PortId_t	Port number of remote L4 protocol.

---

```
bool Close()
```

Close an open connection.

**Return Value:**

Type	Description
bool	True if close is successful. This does not mean that a TCP connection has completely closed, since the return is immediate. TCP will timeout and retry the close action if the FIN packet is lost.

---

```
TTL(Count_t)
```

Set the time-to-live value to set in all packet created by this protocol endpoing.

**Arguments:**

Type	Description
Count_t	TTL value to set.

---

```
Count_t TTL()
```

Return the current TTL value for this connection.

**Return Value:**

Type	Description
Count_t	Current TTL value for this connection.

---

```
FlowId(FId_t)
```

Assign a *Flow Identifier* for this connection. The flow id is a simulation artifact with no basis in real networks, but is useful for simulations to keep track of all packets associated with a given flow. The flow identifier is logged in the trace file for each packet.

**Arguments:**

Type	Description
FId_t	Flow identifier to assign.

---

```
FId_t FlowId()
```

Return the current flow identifier for this connection.

**Return Value:**

Type	Description
FId_t	Flow identifier for this connection.

---

```
L4Protocol* Copy()
```

Make a copy of this protocol endpoint.

**Return Value:**

Type	Description
L4Protocol*	Pointer to copy of this protocol.

---

```
Proto_t ProtocolNumber()
```

Return the protocol number registered for this layer 4 protocol.

**Return Value:**

Type	Description
Proto_t	Protocol number.

---

```
bool IsTCP()
```

Determine if this endpoint is TCP.

**Return Value:**

Type	Description
bool	True if TCP

---

```
AttachApplication(Application*)
```

Attach an application to this L4 protocol.

**Arguments:**

Type	Description
Application*	Pointer to application to attach.

---

```
Application* GetApplication()
```

Get the currently attached application.

**Return Value:**

Type	Description
Application*	Pointer to the currently attached application, or nil if none.

---

```
AddNotification(NotifyHandler*)
```

Adds a "Packet Transmission" notification for all packets sent by this protocol.

**Arguments:**

Type	Description
NotifyHandler*	Notifier object to receive the notification.

---

```
bool IsColored()
```

Determine if this application has a color assigned for created packets

**Return Value:**

Type	Description
bool	True if a color assigned.

---

```
SetColor(const QColor&)
```

Specify a color for all packets generated by this protocol.

**Arguments:**

Type	Description
const QColor&	Desired color (see qcolor.h in qt).

---

```
Packet* NewPacket()
```

Allocate a new packet. Uses either a normal "uncolored" packet or a colored packet, depending on whether the protocol is colored or not.

---

```
AddExtraRxDelay(Time_t)
```

The "Extra Delay" mechanism allows the simulator to vary the round-trip time on a protocol by protocol basis, rather than using link delays. The Rx delay is extra delay added to received packets.

**Arguments:**

Type	Description
Time_t	Extra propagation delay on the last-hop link.

---

```
AddExtraTxDelay(Time_t)
```

The "Extra Delay" mechanism allows the simulator to vary the round-trip time on a protocol by protocol basis, rather than using link delays. The Rx delay is extra delay added to received packets.

**Arguments:**

Type	Description
Time_t	Extra propagation delay on the last-hop link.

---

## 6.2 TCP Protocol Base Class and Variations

Class TCP. Class TCP is the base class for all TCP models. Inherits from L4Protocol.

**TCP Constructors:**

```
TCP()
```

Default constructor, no arguments.

---

TCP(Node\*)

Construct a TCP endpoint attached to a particular node.

**Arguments:**

Type	Description
Node*	Node pointer to attach this TCP protocol to.

---

TCP(const TCP&)

Copy constructor.

**Arguments:**

Type	Description
const TCP&	TCP protocol to copy.

---

**TCP Public Methods:**

bool Listen()

This this endpoint to the listening state to accept incoming connections.

**Return Value:**

Type	Description
bool	True if successful.

---

SetRTTEstimator(const RTTEstimator&)

Set a non-default round trip time estimator.

**Arguments:**

Type	Description
const RTTEstimator&	Reference to new RTT estimator for this TCP.

---

RTTEstimator\* GetRTTEstimator()

Get a pointer to the current RTT estimator.

**Return Value:**

Type	Description
RTTEstimator*	Pointer to current RTT estimator.

---

SetSSThresh(DCount\_t)

Set the slow start threshold for this connection.

**Arguments:**

Type	Description
DCount_t	Slow start threshold (bytes).

---

SetAdvertisedWindow(Count\_t)

Set the receiver advertised window for this connection.

**Arguments:**

Type	Description
Count_t	Advertised window (bytes) for this connection.

---

DeleteOnComplete(bool)

Specify that this connection should automatically delete itself when it closes.

**Arguments:**

Type	Description
bool	True if delete on complete desired. <b>Default Value</b> is true

---

```
DeleteOnTWait(bool)
```

Specify that this connection should automatically delete itself on the completion of the *Timed Wait* state.

**Arguments:**

Type	Description
bool	True if delete on timed wait desired. <b>Default Value</b> is true

---

```
CloseOnEmpty(bool)
```

Specify that this connection should initiate close action when all pending data has been sent.

**Arguments:**

Type	Description
bool	True if close on empty desired. <b>Default Value</b> is true

---

```
SetSegSize(Count_t)
```

Set the segment size for this connection.

**Arguments:**

Type	Description
Count_t	Segment size (bytes).

---

```
SetTxBuffer(Count_t)
```

Set the size of the transmit buffer. Note: Presently not enforced.

**Arguments:**

Type	Description
Count_t	Size of transmit buffer (bytes).

---

```
SetTwTimeout(Time_t)
```

Set the length of the *Timed Wait* state.

**Arguments:**

Type	Description
Time_t	Length of timed wait state (seconds).

---

```
SetConnTimeout(Time_t)
```

Set the length of the connection timeout.

**Arguments:**

Type	Description
Time_t	Length of connection state (seconds).

---

```
SetChildLimit(Count_t)
```

Set a limit on the number of simultaneous child (spawned by a listening endpoint).

**Arguments:**

Type	Description
Count_t	Desired child limit.

---

```
Rate_t GoodPut()
```

Return the current goodput (bits / second) for this connection.

**Return Value:**

Type	Description
Rate_t	Goodput in bits per second.

---

```
Count_t RetransmitCount()
```

Query the number of packet retransmissions for this connection.

**Return Value:**

Type	Description
Count_t	Retransmitted packet count.

---

```
EnableTimeSeq(TimeSeqSelector_t)
```

The TCP model in *GTNets* will keep detailed sequence versus time history information for various counters. This method enables this data collection. This information can later be streamed to a data file with `LogTimeSeq`. More than one data collection can be enabled at a time with multiple calls to `EnableTimeSeq`.

**Arguments:**

Type	Description
<code>TimeSeqSelector_t</code>	Desired level of history connection. Can be one of: <code>LOG_SEQ_TX</code> (sequence number transmitted), <code>LOG_ACK_TX</code> (ack number transmitted), <code>LOG_SEQ_RX</code> (sequence number received), <code>LOG_ACK_RX</code> (ack number received), <code>LOG_CWIN</code> (congestion window size).

---

```
DisableTimeSeq(TimeSeqSelector_t)
```

Stop history data collection for the specified type of sequence number.

**Arguments:**

Type	Description
<code>TimeSeqSelector_t</code>	Desired history level to stop, same levels as in <code>EnableTimeSeq</code> .

---

```
ResetTimeSeq(TimeSeqSelector_t)
```

Reset the collected information for the specified time/sequence collection.

**Arguments:**

Type	Description
<code>TimeSeqSelector_t</code>	Desired history level to reset, same levels as in <code>EnableTimeSeq</code> .

---

```
LogTimeSeq(TimeSeqSelector_t, std::ostream&, Seq_t, Seq_t, char, ')
```

Log previously collected time/sequence data to a file.

**Arguments:**

Type	Description
<code>TimeSeqSelector_t</code>	Desired Time/Sequence type.
<code>std::ostream&amp;</code>	Output stream to log the data.
<code>Seq_t</code>	If non-zero, each sample is divided by this value. <b>Default Value</b> is 0
<code>Seq_t</code>	If non-zero, each sample is modded by this value. <b>Default Value</b> is 0
<code>char</code>	Separator character between time and sequence number <b>Default Value</b> is <code>'</code>
<code>,</code>	

---

```
DCount_t CWin()
```

Query the current value of the congestion window.

**Return Value:**

Type	Description
<code>DCount_t</code>	Congestion window value (bytes, double).

---

```
TCP* Peer()
```

Get a pointer to remote peer endpoint. This information is not always available, since distributed simulations may not have access to the peer object.

**Return Value:**

Type	Description
<code>TCP*</code>	Pointer to peer TCP object. <code>nil</code> if unknown.

---

```
int State()
```

Query the current *state* of this connection.

**Return Value:**

Type	Description
------	-------------

int	Connection state. See <code>tcp.h</code> for a list of the state values.
-----	--

---

```
Count_t TimeoutCount()
```

Query a count of timeouts for this connection.

**Return Value:**

Type	Description
------	-------------

Count_t	Number of timeouts.
---------	---------------------

---

```
Time_t TimeoutDelay()
```

Query the length of the last scheduled timeout.

**Return Value:**

Type	Description
------	-------------

Time_t	Length of last scheduled timeout (seconds).
--------	---

---

**TCP Static Methods:**

```
DefaultSegSize(Count_t)
```

Set the default segment size. The default value is 512 bytes unless a new value is assigned.

**Arguments:**

Type	Description
------	-------------

Count_t	Default segment size (bytes).
---------	-------------------------------

---

```
DefaultAdvWin(Count_t)
```

Set the default advertised window size. value is 65k bytes unless a new value is assigned.

**Arguments:**

Type	Description
------	-------------

Count_t	Default advertised window size (bytes).The default
---------	--

---

```
DefaultSSThresh(DCount_t)
```

Set the default slow-start threshold size. value is 65k bytes unless a new value is assigned.

**Arguments:**

Type	Description
------	-------------

DCount_t	Default slow-size size (bytes).The default
----------	--

---

```
DefaultTxBuffer(Count_t)
```

Set the default transmit buffer size. value is 4Gb unless a new value is assigned.

**Arguments:**

Type	Description
------	-------------

Count_t	Default transmit buffer size (bytes).The default
---------	--

---

```
DefaultRxBuffer(Count_t)
```

Set the default receive buffer size. value is 4Gb unless a new value is assigned.

**Arguments:**

Type	Description
------	-------------

Count_t	Default receive buffer size (bytes).The default
---------	---

---

DefaultTwTimeout(*Time\_t*)

Set the default value of the the timed wait timeout. value is 5 seconds unless a new value is assigned.

**Arguments:**

Type	Description
<i>Time_t</i>	Default time wait value (seconds).The default

---

DefaultConnTimeout(*Time\_t*)

Set the default value of the the connection timeout. value is 6 seconds unless a new value is assigned.

**Arguments:**

Type	Description
<i>Time_t</i>	Default time to wait for syn-ack (seconds).The default

---

LogFlagsText(*bool*)

Specifies that the trace file entries for this connection should show the flags field in text, such as "SYN|ACK". If not specified, flags values are shown in 8 bit hex.

**Arguments:**

Type	Description
<i>bool</i>	True if text flags desired.

---

UseTimerBuckets(*bool*)

Specifies that all timeout timers should be rounded to integral *bucket* values. If specified, timers are rounded to units of 10ms, which is slightly more efficient in the simulator, and is a bit more realistic. If not specified, timers are nano-second accurate.

**Arguments:**

Type	Description
<i>bool</i>	True if timer buckets desired. <b>Default Value</b> is true

---

DefaultConnCount(*Count\_t*)

Specify the default number of connection retries. value is 3 unless a new value is assigned.

**Arguments:**

Type	Description
<i>Count_t</i>	Default connection retry. The default

---

Default(const TCP&)

Specify the default TCP variation to use for all new TCP endpoints. Default value is TCPTahoe unless a new value is specified.

**Arguments:**

Type	Description
const TCP&	Reference to any TCP variant to set as default.

---

TCP& Default()

Return a reference to the default TCP variation.

**Return Value:**

Type	Description
TCP&	Reference to default TCP variant.

---

Class TCPTahoe. Class TCPTahoe defines the behavior of the *Tahoe* variation of *TCP*. The base class functions `DupAck` and `ReTxTimeout` are overridden here, but all other base class *TCP* functions are used.

TCP Tahoe **Public Methods:**

DupAck(const TCPHeader&, Count\_t)

Process a duplicate ack.

**Arguments:**

Type	Description
const TCPHeader&	TCP Header that caused the duplicate ack.
Count_t	Count of duplicates.

---

ReTxTimeout()

Process a retransmit timeout.

---

Class TCP Reno. Class TCP Reno defines the behavior of the *Reno* variation of *TCP*. The base class functions DupAck, ReTxTimeout and NewAck are overridden here, but all other base class TCP functions are used.

TCP Reno **Public Methods:**

DupAck(const TCPHeader&, Count\_t)

Process a duplicate ack.

**Arguments:**

Type	Description
const TCPHeader&	TCP Header that caused the duplicate ack.
Count_t	Count of duplicates.

---

ReTxTimeout()

Process a retransmit timeout.

---

NewAck(Seq)

Process newly acknowledged data.

**Arguments:**

Type	Description
Seq	Sequence number for new ack.

---

Class TCP New Reno. Class TCP New Reno defines the behavior of the *New Reno* variation of *TCP*. The base class functions DupAck, ReTxTimeout and NewAck are overridden here, but all other base class TCP functions are used.

TCP New Reno **Public Methods:**

DupAck(const TCPHeader&, Count\_t)

Process a duplicate ack.

**Arguments:**

Type	Description
const TCPHeader&	TCP Header that caused the duplicate ack.
Count_t	Count of duplicates.

---

ReTxTimeout()

Process a retransmit timeout.

---

NewAck(Seq)

Process newly acknowledged data.

**Arguments:**

Type	Description
Seq	Sequence number for new ack.

---

## 6.3 Round-Trip Time Estimators

Class `RTTEstimator`. Class `RTTEstimator` is a virtual base class which defines the behavior of a round trip time estimator used by TCP.

### `RTTEstimator` Constructors:

`RTTEstimator()`  
`RTTEstimator` constructor, no arguments.

---

`RTTEstimator(Time_t)`  
`RTTEstimator` constructor specifying the initial RTT estimate.

#### Arguments:

Type	Description
<code>Time_t</code>	Initial RTT Estimate.

---

`RTTEstimator(const RTTEstimator&)`  
Copy constructor.

#### Arguments:

Type	Description
<code>const RTTEstimator&amp;</code>	<code>RTTEstimator</code> object to copy.

---

### `RTTEstimator` Public Methods:

`SentSeq(Seq_t, Count_t)`  
Notify the RTT estimator that a particular sequence number was sent.

#### Arguments:

Type	Description
<code>Seq_t</code>	First sequence number in the block of data sent.
<code>Count_t</code>	Length of block sent (bytes).

---

`Time_t AckSeq(Seq_t)`  
Notify the RTT estimator that a particular sequence number has been acknowledged.

#### Return Value:

Type	Description
<code>Time_t</code>	Measured RTT for this sample.

#### Arguments:

Type	Description
<code>Seq_t</code>	Sequence number acknowledged.

---

`ClearSent()`  
Clear the list of pending sequence numbers sent.

---

`Measurement(Time_t)`  
Add a round trip time measurement to the filter.

#### Arguments:

Type	Description
<code>Time_t</code>	Individual measurement (seconds).

---

`Time_t Estimate()`  
Return the current RTT estimate.

#### Return Value:

Type	Description
<code>Time_t</code>	Current RTT estimate (seconds).

---

Time\_t RetransmitTimeout()  
Return the computed retransmit timeout period.

**Return Value:**

Type	Description
Time_t	Retransmit timeout period (seconds).

---

Init(Seq\_t)  
Set the initial sequence number for this estimator.

**Arguments:**

Type	Description
Seq_t	Initial sequence number.

---

RTTEstimator\* Copy()  
Make a copy of this estimator.

**Return Value:**

Type	Description
RTTEstimator*	Pointer to copy of this estimator.

---

IncreaseMultiplier()  
Increase the multiplier for the retransmit timeout calculation. Many TCP implementations exponentially increase the retransmit timeout period on each timeout action.

---

ResetMultiplier()  
Reset the retransmit multiplier to the initial value.

---

Reset()  
Reset this RTT estimator to its initial state.

---

SetPrivateStatistics(const Statistics&)  
Specify a statistics object to track the behavior of this estimator object.

**Arguments:**

Type	Description
const Statistics&	Reference to any statistics object.

---

Statistics\* GetPrivateStatistics()  
Get a pointer to the statistics object for this estimator.

**Return Value:**

Type	Description
Statistics*	Statistics pointer to current private statistics object, nil if none.

---

**RTTEstimator Static Methods:**

InitialEstimate(Time\_t)  
Set a global default initial estimate.

**Arguments:**

Type	Description
Time_t	Initial estimate for all rtt estimators.

---

Default(const RTTEstimator&)

Set a new global default RTT estimator.

**Arguments:**

Type	Description
const RTTEstimator&	Reference to an RTT estimator to use for global default.

---

RTTEstimator\* Default()

Get a pointer to the current default RTT estimator.

**Return Value:**

Type	Description
RTTEstimator*	Pointer to current default RTT estimator.

---

SetStatistics(Statistics\*)

Set a statistics object for global statistics. This collects the statistics from all RTT estimators in a single statistics object.

**Arguments:**

Type	Description
Statistics*	Reference to a statistics object for global data collection.

---

Class RTTMDev. Class RTTMDev implements the "Mean-Deviation" estimator as described by Van Jacobson "Congestion Avoidance and Control", SIGCOMM 88, Appendix A

RTTMDev **Constructors:**

RTTMDev(Mult\_t)

Constructor for RTTMDev specifying the gain factor for the estimator.

**Arguments:**

Type	Description
Mult_t	Gain factor.

---

RTTMDev(Mult\_t, Time\_t)

Constructor for RTTMDev specifying the gain factor and the initial estimate.

**Arguments:**

Type	Description
Mult_t	Gain factor.
Time_t	Initial estimate.

---

RTTMDev(const RTTMDev&)

Copy constructor.

**Arguments:**

Type	Description
const RTTMDev&	RTTMDev object to copy.

---

RTTMDev **Public Methods:**

Gain(Mult\_t)

Set the filter gain for this estimator.

**Arguments:**

Type	Description
Mult_t	Gain factor.

---

## 6.4 UDP Protocol

Class UDP. Class UDP defines a model of the User Datagram Protocol. UDP inherits from L4Protocol and implements most of the methods defined there.

### UDP Constructors:

UDP ( )  
Default constructor, no arguments.

---

UDP (Node\* )  
Constructor, specify a node to attached to.

#### Arguments:

Type	Description
Node*	Node this UPD protocol is attached to.

---

UDP (const UDP& )  
Copy constructor.

#### Arguments:

Type	Description
const UDP&	UDP object to copy.

---

### UDP Public Methods:

PacketSize (Size\_t )  
Set the packet size for this UDP endpoint.

#### Arguments:

Type	Description
Size_t	Packet size (bytes).

---

Size\_t PacketSize ( )

#### Return Value:

Type	Description
Size_t	Query packet size for this UDP endpoint.

---

### UDP Static Methods:

DefaultPacketSize (Size\_t )  
Set default packet size for all UDP protocol objects.

#### Arguments:

Type	Description
Size_t	Default packet size (bytes). Default size is 512 bytes unless otherwise specified.

---

## Chapter 7

# Layer 3 Protocols

Presently, *GTNets* uses *IP* version 4 for all layer 3 processing. The `IPV4` object is described in detail below.

**Class `IPV4`.** The class `IPV4` implements the Internet Protocol Version 4. It derives from the class `L3Protocol`. Since `IPV4` is a static instance of this is enough for a simulation that uses IPv4.

### `IPV4` Constructors:

`IPV4()`  
Default constructor

---

### `IPV4` Public Methods:

`DataRequest(Node*, Packet*, void*)`  
This method is used by layer 4 protocols like `udp/tcp` to hand a packet to IP.

#### Arguments:

Type	Description
<code>Node*</code>	Pointer to the node requesting the data transfer
<code>Packet*</code>	Pointer to the packet to be transmitted
<code>void*</code>	Any auxiliary information that layer 4 might want to convey

---

`DataIndication(Interface*, Packet*)`  
This method is used to pass on a packet from any layer 2 protocol up the stack.

#### Arguments:

Type	Description
<code>Interface*</code>	A pointer to the interface at which this packet is received
<code>Packet*</code>	A pointer to the packet that was received

---

`Interface* PeekInterface(Node*, void*)`  
This method returns the interface pointer. This interface is chosen based on the node pointer and some arbitrary data passed to it.

#### Return Value:

Type	Description
<code>Interface*</code>	the pointer to the interface object

#### Arguments:

Type	Description
<code>Node*</code>	the pointer to the associated node
<code>void*</code>	the data based on which we return the address

---

`Count_t Version()`  
This method returns the version of Internet Protocol

---

Proto\_t Proto()

This method returns the IANA assigned protocol number

**Return Value:**

Type	Description
Proto_t	the protocol number

---

**IPV4 Static Methods:**

IPV4\* Instance()

This method returns the pointer to the global instance of the IPV4 object

**Return Value:**

Type	Description
IPV4*	the pointer to the global instance

---

# Chapter 8

## Layer 2 Protocols

*GTNets* has simulation models for the IEEE 802-3 protocol for wired links, and IEEE 802.11 protocol for wireless. These are described in detail below.

### 8.1 IEEE 802.2

Class `L2Proto802_3`. Class `L2Proto802_3` defines the 802.3 layer 2 protocol. This is derived from class `L2Proto`. It basically implements the Medium Access Control portion of layer 2.

#### `L2Proto802_3` Constructors:

`L2Proto802_3()`  
The Default Constructor

---

#### `L2Proto802_3` Public Methods:

`BuildDataPDU(MACAddr, MACAddr, Packet*)`  
Builds the layer2 pdu and appends it to the specified packet.

##### Arguments:

Type	Description
<code>MACAddr</code>	Source MAC address
<code>MACAddr</code>	Destination MAC address
<code>Packet*</code>	Packet pointer to append this pdu to.

---

`DataRequest(Packet*)`

This is the real member function is called by the higher layer requesting a transfer of data to a peer for 802.3

##### Arguments:

Type	Description
<code>Packet*</code>	The packet to be sent

---

`DataIndication(Packet*)`

This method is called when the direction of protocol stack traversal is upwards, i.e, a packet has just been received at the underlying link

##### Arguments:

Type	Description
<code>Packet*</code>	The packet that has been received

---

bool Busy()

Test if the protocol (or link) is busy processing a packet.

**Return Value:**

Type	Description
bool	True if currently busy.

---

L2Proto802\_3\* Copy()

This method makes a copy of the protocol object and returns the pointer to the copy

**Return Value:**

Type	Description
L2Proto802_3*	pointer to the copy

---

bool IsWireless()

Query if this protocol needs a wireless interface.

**Return Value:**

Type	Description
bool	Always returns false

---

## 8.2 IEEE 802.11

*To Be Supplied*

## Chapter 9

# Routing

All packet level network simulators must have some mechanism to route packets from a source to a destination. One way is to implement in the simulator some well known routing protocols (such as *BGP* or *OSPF*) and use those protocols to build up *routing tables* at each node in the simulated environment. Another approach is to utilize the fact that in a simulator with global topology knowledge, it is easy to simply compute routes a priori using any one of a number of algorithms that statically analyze graphs (such as Dijkstra's shortest path algorithm).

However, any approach that uses routing tables at each node consumes memory in the simulator quadratically as a function of the number of nodes defined. For example, to model a 100 node topology each of the 100 nodes needs a routing table that can contain as many as 100 entries, leading to  $O(n^2)$  memory usage. To alleviate this excessive memory requirement, *GTNets* uses a source-based routing method known as NIX-Vector routing, which bypasses the computation of routing tables. Instead, routes are computed on demand and cached as a source node for later re-use. See *Stateless Routing in Network Simulations* by Riley et. al, Mascots 2000 for details.

In addition, *GTNets* also provides a method to use the routing tables based approach should simulation user need routing tables his own purposes. When creating routing tables, either a *Static* method (where all routes are pre-computed a priori) or a *Manual* method (where routing information is specified by the user) can be used. Finally, for wireless simulations, *GTNets* uses either *Dyanmic Source Routing (DSR)* or a wireless form of NIX-Vector routing known as *NVR*. When creating *GTNets* simulations, normally the only use of a routing object would be to specify the default routing type using `SetRouting` described below.

### 9.1 Routing Base Class

Class `Routing`. The class `Routing` is a virtual base class for all the routing protocols that may be needed for a simulation. Each of these routing protocols have to derive from this class and define the routing protocol procedures

#### `Routing` Constructors:

```
Routing()
```

This method is the default constructor for class `Routing`.

---

#### `Routing` Public Methods:

```
Routing()
```

This is the default destructor. Each routing protocol must define its own destructor.

---

```
Default(RoutingEntry)
```

This method adds the default route to a particular node.

#### Arguments:

Type	Description
<code>RoutingEntry</code>	The default routing entry or the default gateway

---

```
Add(IPAddr_t, Count_t, Interface*, IPAddr_t)
```

This method is used to add a routing entry into the routing table.

**Arguments:**

Type	Description
IPAddr_t	The destination IP Address
Count_t	The subnet Mask of the destination IP Address
Interface*	The interface to use
IPAddr_t	The next hop IP Address

---

```
RoutingEntry Lookup(Node*, IPAddr_t)
```

This method implements the lookup mechanism at a node for a given destination IP address

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
Node*	Node which is looking up the routing table
IPAddr_t	Destination IP address

---

```
RoutingEntry LookupFromPDU(PDU*)
```

This method uses the PDU to look up the next route. Such a mechanism is used in routing protocols like nix vector routing

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
PDU*	The pointer to the PDU

---

```
Routing* Clone()
```

This method is used to make copies of the routing object. Such a mechanism is useful for example, in topology creation

**Return Value:**

Type	Description
Routing*	A pointer to the cloned object

---

```
RType_t Type()
```

This method enables to query the routing object of the routing protocol it implements

**Return Value:**

Type	Description
RType_t	the routing type

---

```
InitializeRoutes(Node*)
```

This method initializes the routes and the routing table at a given node

**Arguments:**

Type	Description
Node*	The pointer to the Node at which the routes are to be initialized

---

```
bool NeedInit()
```

This method determines if initialization is needed and returns a bool

**Return Value:**

Type	Description
bool	true if initialization is needed

---

```
Size_t Size()
```

This method gives the size of the routing table, or the Forwarding Information Base Size of the FIB

---

#### Routing Static Methods:

```
Routing* Default()
```

This method returns the default routing protocol object used by default

##### Return Value:

Type	Description
Routing*	A pointer to the default routing object

---

```
SetRouting(Routing*)
```

This method is used to set a new routing protocol for a simulation

##### Arguments:

Type	Description
Routing*	A pointer to the new routing protocol object

---

## 9.2 Nix-Vector Routing

Class `RoutingNixVector`. The class `RoutingNixVector` derives from class `Routing` and defines the NixVector Routing. This is useful for simulating large topologies with comparatively lesser memory requirements.

#### RoutingNixVector Constructors:

```
RoutingNixVector()
```

Default Constructor

---

#### RoutingNixVector Public Methods:

```
RoutingNixVector()
```

The Destructor

---

```
Default(RoutingEntry)
```

This method adds the default route to a particular node.

##### Arguments:

Type	Description
RoutingEntry	The default routing entry or the default gateway

---

```
Add(IPAddr_t, Count_t, Interface*, IPAddr_t)
```

This method is used to add a routing entry into the routing table.

##### Arguments:

Type	Description
IPAddr_t	The destination IP Address
Count_t	The subnet Mask of the destination IP Address
Interface*	The interface to use
IPAddr_t	The next hop IP Address

---

```
RoutingEntry Lookup(Node*, IPAddr_t)
```

This method implements the lookup mechanism at a node for a given destination IP address

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
Node*	Node which is looking up the routing table
IPAddr_t	Destination IP address

---

```
RoutingEntry LookupFromPDU(PDU*)
```

This method uses the PDU to look up the next route. Such a mechanism is used in routing protocols like nix vector routing

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
PDU*	The pointer to the PDU

---

```
Routing* Clone()
```

This method is used to make copies of the routing object. Such a mechanism is useful for example, in topology creation

**Return Value:**

Type	Description
Routing*	A pointer to the cloned object

---

```
RType_t Type()
```

This method enables to query the routing object of the routing protocol it implements

**Return Value:**

Type	Description
RType_t	the routing type

---

```
InitializeRoutes(Node*)
```

This method initializes the routes and the routing table at a given node

**Arguments:**

Type	Description
Node*	The pointer to the Node at which the routes are to be initialized

---

```
bool NeedInit()
```

This method determines if initialization is needed and returns a bool

**Return Value:**

Type	Description
bool	true if initialization is needed

---

```
Size_t Size()
```

This method gives the size of the routing table, or the Forwarding Information Base Size of the FIB

---

```
NixVectorOption* GetCachedNixVector(IPAddr_t)
```

This method locates a previously cached copy of the nix vector to the specified IP address,

**Return Value:**

Type	Description
NixVectorOption*	the pointer to the NixVectorOption PDU, or nil of none

**Arguments:**

Type	Description
IPAddr_t	The destination address

---

```
NixVectorOption* GetNixVector(Node*, IPAddr_t)
```

This method returns a nix vector options PDU corresponding to an IP address at a specified node

**Return Value:**

Type	Description
NixVectorOption*	the pointer to the NixVectorOption PDU

**Arguments:**

Type	Description
Node*	The node at which we need the Nix Vector
IPAddr_t	The destination address

---

```
DBDump(Node*)
```

This method is for debugging

**Arguments:**

Type	Description
Node*	Pointer to the node at which we need debug info

---

## 9.3 Static Routing

Class `RoutingStatic`. The class `RoutingStatic` defines the static routing class. This routing object calculates all the routes needed at a single instance. Static routing is simple and easy to use by the user, but is quite memory intensive. It is recommended for simulation topologies on the order of a few hundred nodes. Larger topologies should use either `Manual` or `NixVector` routing.

**RoutingStatic Constructors:**

```
RoutingStatic()
The default constructor
```

---

**RoutingStatic Public Methods:**

```
RoutingStatic()
The destructor
```

---

```
Default(RoutingEntry)
This method adds the default route to a particular node.
```

**Arguments:**

Type	Description
RoutingEntry	The default routing entry or the default gateway

---

```
Add(IPAddr_t, Count_t, Interface*, IPAddr_t)
```

This method is used to add a routing entry into the routing table.

**Arguments:**

Type	Description
IPAddr_t	The destination IP Address
Count_t	The subnet Mask of the destination IP Address
Interface*	The interface to use
IPAddr_t	The next hop IP Address

---

```
RoutingEntry Lookup(Node*, IPAddr_t)
```

This method implements the lookup mechanism at a node for a given destination IP address

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
Node*	Node which is looking up the routing table
IPAddr_t	Destination IP address

---

```
RoutingEntry LookupFromPDU(PDU*)
```

This method uses the PDU to look up the next route. Such a mechanism is used in routing protocols like nix vector routing

**Return Value:**

Type	Description
RoutingEntry	The corresponding routing entry or nil

**Arguments:**

Type	Description
PDU*	The pointer to the PDU

---

```
Routing* Clone()
```

This method is used to make copies of the routing object. Such a mechanism is useful for example, in topology creation

**Return Value:**

Type	Description
Routing*	A pointer to the cloned object

---

```
RType_t Type()
```

This method enables to query the routing object of the routing protocol it implements

**Return Value:**

Type	Description
RType_t	the routing type

---

```
InitializeRoutes(Node*)
```

This method initializes the routes and the routing table at a given node

**Arguments:**

Type	Description
Node*	The pointer to the Node at which the routes are to be initialized

---

```
bool NeedInit()
```

This method determines if initialization is needed and returns a bool

**Return Value:**

Type	Description
bool	true if initialization is needed

---

```
Size_t Size()
```

This method gives the size of the routing table. or the Forwarding Information Base Size of the FIB

---

## 9.4 Manual Routing

Class `RoutingManual`. The Class `RoutingManual` describes the behavior of manual routing. Thier is no way here of the routing protocol maintaining the routing table. Instead the user adds the entries explicitly

### `RoutingManual` Constructors:

```
RoutingManual()
```

This method is the default constructor for class `RoutingManual`.

---

### `RoutingManual` Public Methods:

```
RoutingManual()
```

This is the destructor method.

---

```
Default(RoutingEntry)
```

This method adds the default route to a particular node.

#### Arguments:

Type	Description
<code>RoutingEntry</code>	The default routing entry or the default gateway

---

```
Add(IPAddr_t, Count_t, Interface*, IPAddr_t)
```

This method is used to add a routing entry into the routing table.

#### Arguments:

Type	Description
<code>IPAddr_t</code>	The destination IP Address
<code>Count_t</code>	The subnet Mask of the destination IP Address
<code>Interface*</code>	The interface to use
<code>IPAddr_t</code>	The next hop IP Address

---

```
RoutingEntry Lookup(Node*, IPAddr_t)
```

This method implements the lookup mechanism at a node for a given destination IP address

#### Return Value:

Type	Description
<code>RoutingEntry</code>	The corresponding routing entry or nil

#### Arguments:

Type	Description
<code>Node*</code>	Node which is looking up the routing table
<code>IPAddr_t</code>	Destination IP address

---

```
RoutingEntry LookupFromPDU(PDU*)
```

This method uses the PDU to look up the next route. Such a mechanism is used in routing protocols like nix vector routing

#### Return Value:

Type	Description
<code>RoutingEntry</code>	The corresponding routing entry or nil

#### Arguments:

Type	Description
<code>PDU*</code>	The pointer to the PDU

---

---

```
Routing* Clone()
```

This method is used to make copies of the routing object. Such a mechanism is useful for example, in topology creation

**Return Value:**

Type	Description
Routing*	A pointer to the cloned object

---

```
RType_t Type()
```

This method enables to query the routing object of the routing protocol it implements

**Return Value:**

Type	Description
RType_t	the routing type

---

```
InitializeRoutes(Node*)
```

This method initializes the routes and the routing table at a given node

**Arguments:**

Type	Description
Node*	The pointer to the Node at which the routes are to be initialized

---

```
bool NeedInit()
```

This method determines if initialization is needed and returns a bool

**Return Value:**

Type	Description
bool	true if initialization is needed

---

```
Size_t Size()
```

This method gives the size of the routing table. or the Forwarding Information Base Size of the FIB

---

## 9.5 Wireless Dynamic Source Routing

*To be provided*

## Chapter 10

# Tracing Packets

One way to observe the results of a simulated network is to create a *trace file* that indicates the state of each packet as it is forwarded through the network. In its simplest form, a trace file will contain an entry for every single packet transmitted and received on every single link in the network. For small simulations, this approach is acceptable. However, for larger simulations such a method can quickly become unwieldy and result in giga-bytes of traced information.

*GTNets* allows very fine-grained control over what information is logged to a trace file. Packet tracing can be enabled or disabled by protocol layer, individual protocol object, or individual nodes within the topology. Further, within a given protocol, individual data items can be enabled or disabled as needed. For example, one can specify that only the *TCP* header should be traced, and within the *TCP* header only the sequence number, acknowledgment number, and flags are of interest. Using this selective enabling and disabling, a manageable size trace file can be created, even for large simulations.

The trace file is globally enabled or disabled by using the `Trace` object described below. Using the `Open` method for the global trace file object, the tracing is initiated and the file name is specified. The trace file should later be closed using the `Close` method.

The enabling or disabling of tracing at various points in the *GTNets* simulation is controlled by the `SetTrace` method, which is defined for nodes and all protocol objects. The argument to the `SetTrace` method is one of three states:

1. `Trace::DISABLED`. This indicates that the trace entry should *NOT* be created.
2. `Trace::ENABLED`. This indicates that the trace entry *should* be created.
3. `Trace::DEFAULT`. This indicates that the object being queried has no opinion, and the decision should be delegated as described below.

A tracing decision is made whenever a packet is received or transmitted at every protocol layer. Pseudo code for deciding if a protocol header is to be traced is given below:

```
bool CheckTrace(Protocol* proto, Packet* pkt)
  if (NOT {\em trace file open}) return false; // No trace file exists
  Node* n = proto->GetNode(); // Node object where protocol resides
  if (n->TraceStatus() == ENABLED) return true;
  if (n->TraceStatus() == DISABLED) return false;
  // Node status is default, keep checking
  // Next check for global protocol layer enabled/disabled
  if (GlobalTraceStatus(proto->Layer()) == ENABLED) return true;
  if (GlobalTraceStatus(proto->Layer()) == DISABLED) return false;
  // Not set globally, check individual object
  if (proto->TraceStatus() == ENABLED) return true;
  if (proto->TraceStatus() == DISABLED) return false;
  return false;
end CheckTrace;
```

From the above pseudo code, it is clear that all tracing can be enabled or disabled on individual nodes. It may be the case that the only interest is for packets at leaf nodes. In that case, setting the trace status to *DISABLED* for all router nodes will disable tracing at the routers.

A sample trace file excerpt is shown below. All trace file lines start with the simulation time and the node identifier for the node processing the packet. All further information is variable, depending on the trace enabling or disabling for the individual protocol layers and protocol objects. In this example, the layer 4 *TCP* protocol object at node 0 is generating a packet. The fields are:

```

source port (10000),
destination port (80),
sequence number (0),
acknowledgement number (0),
flags (SYN),
Flow ID (0), and
Data Length (0).
```

Note that *TCP* headers do not actually contain a flow id or a data length field. These are both added in the simulation environment as a convenience.

Next is the entry for the layer 3 protocol (*IPV4*). The fields are:

```

TTL (64)
Protocol Number (6)
Source IP Address (192.168.0.1)
Destination IP Address (192.168.1.1)
Packet Unique Identifier (1)
```

The next few lines show the packet being routed through the network to the destination at node id 4. For nodes 2 and 3 only the *IP* header is traced, since the packet was processed by the *IP* layer only at these nodes. For this example, all layer 2 tracing is disabled. The packet arrives at the destination node (4) at time 1.88897, and is processed by the *TCP* protocol at that node. This protocol creates a *SYN|ACK* packet and forwards it back to node 0.

This example is typical, but the individual data items being traced at each protocol layer is fully configurable by the user, so the actual trace files may be different.

## 10.1 Trace File Object

Class `Trace`. Class `Trace` defines the behavior of the packet tracing feature of *GTNets*.

### Trace Constructors:

```

Trace()
Default constructor, no arguments.
```

---

### Trace Public Methods:

```

bool Open(const char*)
Create a trace file.
```

#### Return Value:

Type	Description
bool	//Doc:Arg1 File name. True if successfully opened.

#### Arguments:

Type	Description
const char*	

```

1.77891 N0 L4-TCP 10000 80 0 0 SYN 0 0 L3-4 64 6 192.168.0.1 192.168.1.1 1
1.78391 N2 L3-4 63 6 192.168.0.1 192.168.1.1 1
1.88396 N3 L3-4 62 6 192.168.0.1 192.168.1.1 1
1.88897 N4 L3-4 61 6 192.168.0.1 192.168.1.1 1 L4-TCP 10000 80 0 0 SYN 0 0
1.88897 N4 L4-TCP 80 10000 0 0 SYN|ACK 0 0 L3-4 64 6 192.168.1.1 192.168.0.1 2
1.89398 N3 L3-4 63 6 192.168.1.1 192.168.0.1 2
1.99403 N2 L3-4 62 6 192.168.1.1 192.168.0.1 2
1.99904 N0 L3-4 61 6 192.168.1.1 192.168.0.1 2 L4-TCP 80 10000 0 0 SYN|ACK 0 0
1.99904 N0 L4-TCP 10000 80 0 0 ACK 0 0 L3-4 64 6 192.168.0.1 192.168.1.1 4
1.99904 N0 L4-TCP 10000 80 0 0 0 0 512 L3-4 64 6 192.168.0.1 192.168.1.1 5
2.00404 N2 L3-4 63 6 192.168.0.1 192.168.1.1 4
2.00409 N2 L3-4 63 6 192.168.0.1 192.168.1.1 5
2.10409 N3 L3-4 62 6 192.168.0.1 192.168.1.1 4
2.10456 N3 L3-4 62 6 192.168.0.1 192.168.1.1 5
2.10910 N4 L3-4 61 6 192.168.0.1 192.168.1.1 4 L4-TCP 10000 80 0 0 ACK 0 0
2.10961 N4 L3-4 61 6 192.168.0.1 192.168.1.1 5 L4-TCP 10000 80 0 0 0 0 512
2.10961 N4 L4-TCP 80 10000 0 512 ACK 0 0 L3-4 64 6 192.168.1.1 192.168.0.1 7

```

Figure 10.1: GTNets Log File Excerpt

---

Close()

Close the trace file. Should be called when the simulation completes.

---

On()

Enable tracing for all protocol layers.

---

On(Layer\_t)

Enable tracing for a specified protocol layer.

**Arguments:**

Type	Description
Layer_t	Layer to enable (2, 3, or 4).

---

Off()

Enable tracing for all protocol layers.

---

Off(Layer\_t)

Disable tracing for a specified protocol layer.

**Arguments:**

Type	Description
Layer_t	Layer to disable (2, 3, or 4).

---

bool IsOn(Layer\_t)

Test whether tracing is on or off by layer.

**Return Value:**

Type	Description
bool	True if tracing enabled for this layer.

**Arguments:**

Type	Description
Layer_t	Layer to test.

---

```
bool IsEnabled()
```

Check if trace file globally enabled.

**Return Value:**

Type	Description
bool	True if enabled.

---

```
IPDotted(bool)
```

Specify that all *IP Addresss* are to be logged in dotted notation. If not, trace in 32 bit hex.

**Arguments:**

Type	Description
bool	True if dotted desired. <b>Default Value</b> is true

---

```
TimePrecision(Count_t)
```

Specify the digits of precision in the time field for the trace file.

**Arguments:**

Type	Description
Count_t	Number of digits of precision.

---

**Trace Static Methods:**

```
Trace* Instance()
```

Get a pointer to the single, global trace object.

**Return Value:**

Type	Description
Trace*	Pointer to trace object.

---

# Chapter 11

## Miscellaneous Support Objects

*GTNets* provides objects that support a number of services that are typically needed by a simulation. These include data collection, random numbers, *IP Address* processing methods, and protocol data unit (PDU) definitions, among others.

### 11.1 Random Number Generators

*GTNets* provides a number of different ways to generate random numbers. All of the random number generation is based on a well known uniform generator by Pierre L'Ecuyer at the University of Montreal, for which we are grateful. The L'Ecuyer generator uses a seed consisting of 6 integer values of 32 bits each. *GTNets* allows specification of the 6 seeds, a single value replicated 6 times, or a random seed based on the time of day.

#### 11.1.1 Random Number Base Class

Class `Random`. *GTNets* has a rich set of random number generators. Class `Random` defines the base class functionality required for all random number generators. Note: The underlying random number generation method used by *GTNets* is the `RngStream` code by Pierre L'Ecuyer at the University of Montreal.

##### Random Constructors:

`Random()`

Constructor for a random number generator with a random seed.

---

`Random(Seed_t)`

Constructor for a random number generator with a specified seed. The random numbers used by *GTNets* actually use a seed value consisting of 6 32 bit values. The specified seed is replicated six times to create the seed.

##### Arguments:

Type	Description
<code>Seed_t</code>	Seed value to use (32 bit unsigned)

---

##### Random Public Methods:

`Random_t Value()`

Return a floating point random value

##### Return Value:

Type	Description
<code>Random_t</code>	Floating point random value.

---

```
IRandom_t IntValue()
```

Return an integer random value.

**Return Value:**

Type	Description
IRandom_t	Integer random value.

---

```
Seed(Seed_t)
```

Specify a seed for the random number generators. If the specified value is zero, then the RNG is seeded randomly by one of three methods. First, if a global seed random number generator is specified (See static method `GlobalSeed`) then the seed RNG is sampled for the seed. If `UseDevRandom` is specified, then the linux device `/dev/random` is sampled for the seed. If neither of these is specified, then the time of day is used as a random seed. If the specified value is not zero, the six copies of this are used for the seed.

**Arguments:**

Type	Description
Seed_t	Random seed to use.

---

```
Seeds(Seed_t*)
```

Seed the random number generator with six seeds.

**Arguments:**

Type	Description
Seed_t*	Pointer to an array of six 32 bit unsigned to use for seeds.

---

**Random Static Methods:**

```
UseDevRandom(bool)
```

Specify whether the linux device `/dev/random` is to be used for a random seed.

**Arguments:**

Type	Description
bool	True if <code>/dev/random</code> desired. <b>Default Value</b> is true

---

```
GlobalSeed(Seed_t, Seed_t, Seed_t, Seed_t, Seed_t, Seed_t)
```

It is often desirable to create a simulation that uses random numbers, while at the same time is completely reproducible. Specifying this set of six random seeds creates a special random number generator used for all random seeds for other random number generators. Thus, specifying the `tt RANDOM_SEED` value for the seed for each created generator will result in a known and reproducible set of seeds, and a reproducible set of samples.

**Arguments:**

Type	Description
Seed_t	Seed to use (32 bit unsigned)
Seed_t	Seed to use (32 bit unsigned)
Seed_t	Seed to use (32 bit unsigned)
Seed_t	Seed to use (32 bit unsigned)
Seed_t	Seed to use (32 bit unsigned)
Seed_t	Seed to use (32 bit unsigned)

---

### 11.1.2 Uniform Random

Class `Uniform`. Creates a uniformly distributed random number generator.

**Uniform Constructors:**

Uniform()

Creates a uniform random number generator in the range [0.0 .. 1.0)

---

Uniform(Random\_t, Random\_t)

Creates a uniform random number generator with the specified range.

**Arguments:**

Type	Description
Random_t	
Random_t	

---

**11.1.3 Constant Random**

Class `Constant`. Class `Constant` defines a random number generator that returns the same value every sample. It's not immediately obvious why such a generator is useful, but it is. A number of other *GTNets* objects require a random number generator as parameters to describe the behavior. For example, application `TCPSend` uses a random number generator to determine how much data to send. By passing a `Constant` RNG to the `TCPSend` application, it will send a known constant amount.

**Constant Constructors:**

Constant()

Construct a `Constant` RNG that return zero every sample.

---

Constant(Random\_t)

Construct a `Constant` RNG that returns the specified value every sample.

**Arguments:**

Type	Description
Random_t	Constant value for this RNG.

---

**Constant Public Methods:**

NewConstant(Random\_t)

Specify a new constant RNG for this generator.

**Arguments:**

Type	Description
Random_t	Constant value for this RNG.

---

**11.1.4 Sequential Random**

Class `Sequential`. Class `Sequential` defines a random number generator that returns a sequential sequence.

**Sequential Constructors:**

Sequential(Random\_t, Random\_t, Random\_t, Count\_t)

Constructor for the `Sequential` RNG. The four parameters define the sequence. For example `Sequential(0, 5, 1, 2)` returns the sequence 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 0, 0...

**Arguments:**

Type	Description
Random_t	First value returned.
Random_t	Last + 1 value returned.
Random_t	Increment. <b>Default Value</b> is 1
Count_t	Repeat count for each value. <b>Default Value</b> is 1

---

```
Sequential(Random_t, Random_t, const Random&, Count_t)
```

Constructor for the `Sequential` RNG. Differs from the first constructor only in the fact that the increment parameter is a random variable.

**Arguments:**

Type	Description
<code>Random_t</code>	First value returned.
<code>Random_t</code>	Last + 1 value returned.
<code>const Random&amp;</code>	Reference to a random variable for the sequence increment.
<code>Count_t</code>	Repeat count for each value. <b>Default Value</b> is 1

---

### 11.1.5 Exponential Random

Class `Exponential`. Class `Exponential` defines a random variable with an exponential distribution.

**Exponential Constructors:**

```
Exponential()
```

Constructs an exponential random variable with a mean value of 1.0.

---

```
Exponential(Random_t)
```

Constructs an exponential random variable with the specified mean value.

**Arguments:**

Type	Description
<code>Random_t</code>	Mean value for the random variable.

---

```
Exponential(Random_t, Random_t)
```

Constructs an exponential random variable with the specified mean value and specified upper limit. Since exponential distributions can theoretically return unbounded values, it is sometimes useful to specify a fixed upper limit. Note however that when the upper limit is specified, the true mean of the distribution is slightly smaller than the mean value specified.

**Arguments:**

Type	Description
<code>Random_t</code>	Mean value for the random variable.
<code>Random_t</code>	Upper limit on returned values.

---

### 11.1.6 Pareto Random

Class `Pareto`. Class `Pareto` defines a random variable with a pareto distribution.

**Pareto Constructors:**

```
Pareto()
```

Constructs a pareto random variable with a mean value of 1.0 and a shape (alpha) parameter of 1.5.

---

```
Pareto(Random_t)
```

Constructs a pareto random variable with the specified mean value and a shape (alpha) parameter of 1.5.

**Arguments:**

Type	Description
<code>Random_t</code>	Mean value for the distribution.

---

```
Pareto(Random_t, Random_t)
```

Constructs a pareto random variable with the specified mean value and a shape (alpha) parameter of 1.5.

**Arguments:**

Type	Description
Random_t	Mean value for the distribution.
Random_t	Shape (alpha) parameter for the distribution.

---

```
Pareto(Random_t, Random_t, Random_t)
```

Constructs a pareto random variable with the specified mean value, shape (alpha) parameter of 1.5, and upper bound. Since pareto distributions can theoretically return unbounded values, it is sometimes useful to specify a fixed upper limit. Note however that when the upper limit is specified, the true mean of the distribution is slightly smaller than the mean value specified.

**Arguments:**

Type	Description
Random_t	Mean value for the random variable.
Random_t	Shape (alpha) parameter for the distribution.
Random_t	Upper limit on returned values.

---

### 11.1.7 Empirical Random

Class `Empirical`. Defines a random variable that has a specified, empirical distribution. The distribution is specified by a series of calls to the `CDF` member function, specifying a value and the probability that the function value is less than the specified value. When values are requested, a uniform random variable is used to select a probability, and the return value is interpreted linearly between the two appropriate points in the CDF.

**Empirical Constructors:**

```
Empirical(Seed_t)
```

Constructor for the `Empirical` random variables.

**Arguments:**

Type	Description
Seed_t	Optional seed value to use for selecting points. <b>Default Value</b> is <code>RANDOM_SEED</code>

---

**Empirical Public Methods:**

```
CDF(Random_t, CDF_t)
```

Specifies a point in the empirical distribution.

**Arguments:**

Type	Description
Random_t	The function value for this point.
CDF_t	Probability that the function is less than or equal to the specified value.

---

### 11.1.8 Integer Empirical Random

Class `IntEmpirical`. Defines an empirical distribution where all values are integers. Identical to `Empirical`, but with slightly different interpolation between points.

**IntEmpirical Constructors:**

```
IntEmpirical(Seed_t)
```

Constructor for `IntEmpirical`

**Arguments:**

Type	Description
Seed_t	Optional seed value to use for selecting points. <b>Default Value</b> is <code>RANDOM_SEED</code>

---

## 11.2 Statistics Gathering

A number the *GTNets* objects have an optional statistics gathering object. For example, the web browser object will optionally log on the statistics object the measured web response time for each web object received. These statistics gathering objects can then create a data file that can be used as a spreadsheet input, or as input to a graphing program.

### 11.2.1 Statistics Base Class

Class `Statistics`. Class `Statistics` defines a virtual base class that defines the behavior of all statistics collection objects in *GTNets*.

#### `Statistics` Public Methods:

`bool Record(double)`  
Record a new statistics sample.

#### Arguments:

Type	Description
double	Sample to record.

---

`Reset()`

Remove all samples from the statistics collector and restore initial conditions.

---

`Log(std::ostream&, const char*, const char, ')`

Record the statistics on an output stream.

#### Arguments:

Type	Description
<code>std::ostream&amp;</code>	Output stream to record the statistics on.
<code>const char*</code>	Header line to add to beginning of the statistics log (if not nil). <b>Default Value</b> is nil
<code>const char</code>	Separator character between statistics elements. <b>Default Value</b> is <code>'</code>

---

### 11.2.2 Histogram Class

Class `Histogram`. Class `Histogram` defines a statistics collection object that tracks samples in fixed size buckets and counts the number of samples falling in each bucket interval.

#### `Histogram` Constructors:

`Histogram()`

Default constructor, no arguments. Not particularly useful, since the default constructor sets the maximum measurement to zero, resulting in all measurements being out of range.

---

`Histogram(double)`

Construct with specified maximum measurement.

#### Arguments:

Type	Description
double	Maximum measurement.

---

`Histogram(double, Count_t)`

Construct with specified maximum value and bin count. This is the preferred constructor for Histograms.

#### Arguments:

Type	Description
double	Maximum measurement.
<code>Count_t</code>	Desired bin count.

---

**Histogram Public Methods:**

Bins(double, Count\_t)  
Specify the maximum value and bin count.

**Arguments:**

Type	Description
double	Maximum measurement.
Count_t	Desired bin count.

---

Log(std::ostream&, const char\*, const char, ' )  
Record the statistics on an output stream. The histogram is recorded with two values per line, the minimum value for a bin, and the count of measurements in that bin.

**Arguments:**

Type	Description
std::ostream&	Output stream to record the statistics on.
const char*	Header line to add to beginning of the statistics log (if not nil). <b>Default Value</b> is NULL
const char	Separator character between statistics elements. <b>Default Value</b> is ' , '

---

CDF(std::ostream&, const char\*, const char, ' )  
Record the statistics on an output stream, in Cumulative Distribution Function (CDF) form. The CDF is recorded with two values per line, the maximum value for a bin, and the cumulative probability that the measurement is less than that value.

**Arguments:**

Type	Description
std::ostream&	Output stream to record the statistics on.
const char*	Header line to add to beginning of the statistics log (if not nil). <b>Default Value</b> is NULL
const char	Separator character between statistics elements. <b>Default Value</b> is ' , '

---

**11.2.3 Average–Min–Max Class**

Class AverageMinMax. Class AverageMinMax defines a statistics collection object that dynamically tracks the average, minimum, and maximum values for the measured statistics.

**AverageMinMax Constructors:**

AverageMinMax()  
Default constructor, no arguments.

---

**AverageMinMax Public Methods:**

Count\_t Samples()  
Get a count of the number of sampled measurements.

**Return Value:**

Type	Description
Count_t	Count of number of samples.

---

## 11.3 Packets and PDUs

Packets in *GTNets* consist of a small number of variables to identify the packet and its contents, and a list of associated *Protocol Data Units (PDUs)*. There is one *PDU* for each protocol layer that processed the packet, including optional *Data Contents*.

### 11.3.1 Packets

Class `Packet`. Class `Packet` defines a generic packet class which is composed of a vector of protocol data units (PDUs). It also provides a flexible interface to convert a packet into a serialized buffer for transmission over a network. ( to be used in distributed simulations )

#### Packet Constructors:

`Packet()`  
Default constructor

---

`Packet(const Packet&)`  
This constructor takes a reference to a packet and constructs a new packet with identical contents.

#### Arguments:

Type	Description
<code>const Packet&amp;</code>	reference to a packet p

---

`Packet(char*, Size_t)`  
This constructor constructs a packet object out of the contents of a serialized character buffer of a given size.

#### Arguments:

Type	Description
<code>char*</code>	pointer to the character buffer
<code>Size_t</code>	size of the buffer

---

#### Packet Public Methods:

`Size_t Size()`  
This method returns the size of the packet contents. It adds the contents of all its PDUs and returns the sum.

#### Return Value:

Type	Description
<code>Size_t</code>	the sum

---

`PushPDU(PDU*)`

#### Arguments:

Type	Description
<code>PDU*</code>	

---

`PushPDUBottom(PDU*)`  
This method is used to inset a protocol data unit (PDU) into the packet at the bottom of the stack. The PDU is added at the beginning of the PDU vector.

#### Arguments:

Type	Description
<code>PDU*</code>	pointer to the pdu object

---

PDU\* PopPDU()

---

PDU\* PeekPDU()

---

PDU\* PeekPDU(NCount\_t)

**Arguments:**

Type	Description
NCount_t	

---

SkipPDU()

---

PDU\* PushExisting()

---

PDU\* FindPDU(Layer\_t)

Finds a pdu in the pdu list for the specified layer.

**Return Value:**

Type	Description
PDU*	PDU for specified layer, nil if none

**Arguments:**

Type	Description
Layer_t	Layer to find.

---

PDU\* FindPDU(Layer\_t, Proto\_t)

Finds a pdu in the pdu list for the specified layer and protocol

**Return Value:**

Type	Description
PDU*	PDU for specified layer and protocol, nil if none

**Arguments:**

Type	Description
Layer_t	Layer to find. Protocol to find.
Proto_t	

---

Packet\* Copy()

This method is a generic copy function that makes a copy of itself and returns the pointer to the copy.

**Return Value:**

Type	Description
Packet*	pointer to the copy of the packet

---

Size\_t SSize()

This method returns the total size of the payload in this that needs to be serialized for transmisison over the network to a remote simulator instance. ( for distributed simulations )

**Return Value:**

Type	Description
Size_t	Payload size

---

```
char* Serialize(char*, Size_t&)
```

This method serializes the contents of this packet into a serial character buffer

**Return Value:**

Type	Description
char*	the serialized buffer

**Arguments:**

Type	Description
char*	charater pointer to the buffer
Size_t&	reference to the size of the buffer

---

```
char* Construct(char*, Size_t&)
```

This method does the reverse job of constructing the packet from a character buffer of a give size

**Return Value:**

Type	Description
char*	the pointer to the buffer

**Arguments:**

Type	Description
char*	the serialized character buffer
Size_t&	the size of the serialized buffer

---

```
bool IsColored()
```

Determine if a packet has an associated color (for animation). We define this and the three color component methods (below) as virtual, and always return `false` here. There is a subclass called `ColoredPacket` that will define these values. This is to save memory in packet definitions, as only those packets assigned will have memory to store it.

**Return Value:**

Type	Description
bool	True if colored.

---

```
Color_t R()
```

Return the Red component of the color.

**Return Value:**

Type	Description
Color_t	The Red component.

---

```
Color_t G()
```

Return the Green component of the color.

**Return Value:**

Type	Description
Color_t	The Green component.

---

```
Color_t B()
```

Return the Blue component of the color.

**Return Value:**

Type	Description
Color_t	The Blue component.

---

Packet **Public Members:**

Type	Name	Description
PDUVec_t	PDUs	A vector of protocol data unit (PDU) pointers for this packet.
PDUVec_t::size_type	top	The vector containing the PDU's does not shrink when a PDU is removed, so we need a value specifying what is the index of the current <i>top</i> of the stack.
Count_t	retx	Counter specifying how many times this packet was retransmitted
Count_t	uid	Every packet in <i>GTNets</i> has a unique 32 bit identifier, allowing the packet to be easily identified in the trace file.
Size_t	size	Total size of this packet. This size is measured in number of bytes that will be transmitted on a link for this packet. In <i>GTNets</i> , the PDU representing an IPV4 header (for example) is substantially larger than the 20 bytes in a normal IPV4 PDU. But the size for the IPV4 PDU is claimed to be 20 bytes for purposes of computing transmission time on a link.
Time_t	time	Timestamp for when this packet was originally created.
NixVectorOption*	nixVec	Pointer to associated NIX-Vector for this packet (if present).

Packet **Static Members:**

Type	Name	Description
Count_t	nextUid	Next available unique identifier for a packet.

### 11.3.2 Protocol Data Units

Class PDU. Class PDU serves as the base class for all the protocol data units. Protocol headers and data chunks that form the packets are derived from this class.

PDU **Constructors:**

PDU()

This method is the default constructor for this class. It updates a static variable to reflect the number of PDUs allocated

PDU(const PDU&)

This constructs a PDU from the reference to an already existing PDU.

**Arguments:**

Type	Description
const PDU&	The const reference to the existing PDU

PDU **Public Methods:**

PDU()

Size\_t Size()

This method returns the size of the PDU

**Return Value:**

Type	Description
Size_t	the size of the PDU

---

PDU\* Copy()

This method creates a copy of this PDU

**Return Value:**

Type	Description
PDU*	A pointer to the copy of this PDU

---

Layer\_t Layer()

Each PDU must also return a layer numer as to which protocol layer it belongs to according to the ISO model

**Return Value:**

Type	Description
Layer_t	the layer number

---

Count\_t Version()

If we are using multiple versions of the same PDU we may want to return the version number of the PDU we are using. Also useful when tracing

**Return Value:**

Type	Description
Count_t	the version number

---

Proto\_t Proto()

Most of the protocols are identified by the numeric values as listed by IANA. this method returns that protocol number

**Return Value:**

Type	Description
Proto_t	the protocol number

---

Trace(Tfstream&, Bitmap\_t)

This method traces the contents of the PDU

**Arguments:**

Type	Description
Tfstream&	File output stream descriptor
Bitmap_t	Bitmap that specifies the level of detail of the trace

---

bool Detail(Bitmap\_t, Bitmap\_t)

**Arguments:**

Type	Description
Bitmap_t	
Bitmap_t	

---

Size\_t SSize()

The actual size of all the PDU contents in terms of bytes for serialization of the contents.

**Return Value:**

Type	Description
Size_t	the size

---

```
char* Serialize(char*, Size_t&)
```

This method is used to serialize the contents of the PDU into a serialized character buffer `b` . the size of this buffer is in the size argument

**Return Value:**

Type	Description
char*	the pointer to the updated character buffer

**Arguments:**

Type	Description
char*	This is the pointer to the character buffer
Size_t&	a reference to the size variable

---

```
char* Construct(char*, Size_t&)
```

This method is the reverse of the `Serialize` . It constructs the PDU object from the character buffer that has been received over the network

**Return Value:**

Type	Description
char*	the pointer to the character buffer of the remaining data

**Arguments:**

Type	Description
char*	This is the pointer to the character buffer
Size_t&	a reference to the size variable

---

## 11.4 Command Line Argument Processing

It is often the case that a single simulation will be repeatedly executed, using variations on certain parameters controlling what the simulation does. For example, a web browsing experiment may be run a number of different times with differing values for queue limits at bottleneck queues. *GTNets* provides a simple method for command line argument processing. The code snippet show in figure 11-1 demonstrates the use of the `ARG` parameter processing.

```

1 // Demonstrate use of the GTNetS argument processing
2 // George F. Riley, Georgia Tech, Summer 2003
3
4 #include <iostream>
5 #include "args.h"
6
7 unsigned long pktSize;
8 double        rate;
9
10 // Two command line parameters are defined in this case,
11 // "size" and "rate".  The default size is 1000 and the
12 // default rate is 1e6.  Variable pktSize will default
13 // to 1000, but can be overridden with a command line
14 // parameter, as follows:
15 // ./myGTNetS size=1500 rate=1e7 static
16
17 int main()
18 {
19     Arg("size", pktSize, 1000);
20     Arg("rate", rate, 1e6);
21
22     Arg::ProcessArgs();
23     if (Arg::Specified("static")) {
24         std::cout << "static is specified" << std::endl;
25     }
26     std::cout << "pktSize is " << pktSize << std::endl;
27     std::cout << "rate is " << rate << std::endl;
28 }

```

Program 11-1 argsdemo.cc

Class `Arg`. Since it is often the case that a given simulation scenario must be run multiple times with a variety of different parameters, *GTNets* provides a simple interface to handle command line arguments. Class `Arg` is used to specify an allowable set of equivalenced parameters, which provide default values for parameters as well as explicitly specified value. Further, the `Arg` interface allows for non-equivalenced parameters, which can be queried at runtime for presence or absence.

**Arg Constructors:**

```
Arg(const std::string&, long&, long)
```

Add an argument of type `long`.

**Arguments:**

Type	Description
<code>const std::string&amp;</code>	Argument keyword.
<code>long&amp;</code>	Reference to <code>long</code> variable to receive equivalenced value.
<code>long</code>	Default value if not specified.

```
Arg(const std::string&, unsigned long&, unsigned long)
```

Add an argument of type `unsigned long`.

**Arguments:**

Type	Description
<code>const std::string&amp;</code>	Argument keyword.
<code>unsigned long&amp;</code>	Reference to <code>unsigned long</code> variable to receive equivalenced value.
<code>unsigned long</code>	Default value if not specified.

---

Arg(const std::string&, double&, double)  
Add an argument of type double.

**Arguments:**

Type	Description
const std::string&	Argument keyword.
double&	Reference to double variable to receive equivalenced value.
double	Default value if not specified.

---

Arg(const std::string&, std::string&, const std::string&)  
Add an argument of type string.

**Arguments:**

Type	Description
const std::string&	Argument keyword.
std::string&	Reference to string variable to receive equivalenced value.
const std::string&	Default value if not specified.

---

**Arg Static Methods:**

bool ProcessArgs(int, char\*\*)  
Process the command line arguments. Must be called before checking any argument values.

**Return Value:**

Type	Description
bool	True if successfully processed.

**Arguments:**

Type	Description
int	argc argument from main.
char**	argv argument from main.

---

StringVec\_t::size\_type NumberNonEquiv()  
Return the number of non-equivalenced arguments.

**Return Value:**

Type	Description
StringVec_t::size_type	Count of non-equivalenced argument.

---

std::string NonEquiv(StringVec\_t::size\_type)  
Return the specified non-equivalenced argument.

**Return Value:**

Type	Description
std::string	The specified non-equivalenced argument.

**Arguments:**

Type	Description
StringVec_t::size_type	Index from the non-equivalenced list to return.

---

bool Specified(const std::string&)  
Determine if a particular non-equivalenced argument was specified.

**Return Value:**

Type	Description
bool	True if the argument was specified.

**Arguments:**

Type	Description
const std::string&	

---

## 11.5 IP Address Management Object

Interface objects in *GTNets* can have an optional *IP Address* assigned, much like real interfaces in networks. In order to simplify the specification and management of *IP Addresses*, *GTNets* defines a class `IPAddr` that allows easy manipulation of *IP Address* values.

Class `IPAddr`. Defines an object that specifies an *IP Address* using the familiar "dotted" location. Objects of class `IPAddr` have an automatic typecast to type `IPAddr_t`, so an `IPAddr` object can be used whenever a variable of `IPAddr_t` is needed.

### IPAddr Constructors:

```
IPAddr()
```

Constructor for `IPAddr`. The *IP Address* defaults to `IPADDR_NONE` (0).

```
IPAddr(std::string&)
```

Construct an `IPAddr` object using a string in the familiar "dotted" notation.

#### Arguments:

Type	Description
<code>std::string&amp;</code>	String value specifying the desired <i>IP Address</i> in dotted notation.

```
IPAddr(IPAddr_t)
```

Construct an `IPAddr` object with the specified 32 bit address.

#### Arguments:

Type	Description
<code>IPAddr_t</code>	<i>IP Address</i> as a 32 bit unsigned quantity.

### IPAddr Static Methods:

```
IPAddr_t ToIP(const std::string)
```

Convert a string (dotted notation) *IP Address* to 32 bit unsigned.

#### Return Value:

Type	Description
<code>IPAddr_t</code>	32 bit unsigned <i>IP Address</i> .

#### Arguments:

Type	Description
<code>const std::string</code>	String to convert.

```
char* ToDotted(IPAddr_t)
```

Convert a 32 bit *IP Address* to a string in dotted notation. *IP Address* to convert.

#### Return Value:

Type	Description
<code>char*</code>	The buffer pointed to is only valid until another call to <code>IPAddr</code> functions.

#### Arguments:

Type	Description
<code>IPAddr_t</code>	Pointer to ASCIIZ string with the <i>IP Address</i> in dotted notation.

## 11.6 Time and Rate Parsing Objects

The specification of a time value (such as 10 milliseconds) or a rate value (such as 100 mega-bits per second) is quite common when creating a *GTNets* simulation. Since these values are often thought of or specified in common units (milliseconds, kilo-bits), *GTNets* provides an easy way to specify time and rate values using familiar notation.

### 11.6.1 Time Parsing Object

Class `Time`. Defines an object that specifies time values with familiar suffixes such as "ms". Objects of class `Time` have an automatic typecast define to type `Time_t`, so a `Time` object can be used anywhere a `Time_t` variable is needed.

#### Time Constructors:

```
Time(const std::string)
```

Construct a `Time` object specifying a time as a string.

#### Arguments:

Type	Description
<code>const std::string</code>	String specifying the time. The times can have the suffixes "h" (hours), "m" (minutes), "s" (seconds), "sec" (seconds) "ms" (milliseconds), "us" (microseconds) "ns" (nanoseconds)

---

### 11.6.2 Rate Parsing Object

Class `Rate`. Defines an object that can be used to define rates (bits / second) in familiar units, such as 10Mb. Objects of class `Rate` have an automatic typecast define to type `Rate_t`, so a `Rate` object can be used anywhere a `Rate_t` variable is needed.

#### Rate Constructors:

```
Rate(const std::string)
```

Construct a `Rate` object specifying a rate as a string.

#### Arguments:

Type	Description
<code>const std::string</code>	String specifying the rate. The rates can have the suffixes "b" (bits), "B" (bytes), "Kb" (kilobits), "KB" (kilobytes) "Mb" (megabits), "MB" (megabytes) "Gb" (gigabits), "GB" (gigabytes).

---

Class Name	Include File	Hierarchy
AppOOEvent	application-onoff.h	Event
AppTCPReceiveEvent	application-tcpreceive.h	Event
AppWebBrowserEvent	application-webbrowser.h	Event
Application	application.h	Handler Object
Arg	args.h	
AverageMinMax	average-min-max.h	Statistics
BitMap	bitmap.h	
BrowserConnection	application-webbrowser.h	
CBRAApplication	application-cbr.h	Application
Constant	rng.h	Random
Data	datapdu.h	PDU
DropPDU	droppdu.h	PDU
DropTail	droptail.h	Queue
DropTailIPA	droptail-ipa.h	DropTail Timer
DualQueue	dualqueue.h	Queue
Dumbbell	dumbbell.h	
DuplexLink	duplexlink.h	
Empirical	rng.h	Random
Ethernet	ethernet.h	Link
Event	event.h	ReuseBase
Exponential	rng.h	Random
FTPAction	ftp-client.h	
FTPClient	ftp-client.h	Application TimerHandler
GnuPeerInfo	application-gnutella.h	
Gnutella	application-gnutella.h	Application
Grid	grid.h	
HTTPInfo	application-webserver.h	
Handler	handler.h	
Histogram	histogram.h	Statistics
HttpConsecutivePages	http-distributions.h	IntEmpirical
HttpFilesPerPage	http-distributions.h	IntEmpirical
HttpPrimaryReply	http-distributions.h	IntEmpirical
HttpPrimaryRequest	http-distributions.h	IntEmpirical
HttpSecondaryReply	http-distributions.h	IntEmpirical
HttpSecondaryRequest	http-distributions.h	IntEmpirical
HttpThinkTime	http-distributions.h	Empirical
IPAddr	ipaddr.h	
IPV4	ipv4.h	L3Protocol
IPV4Header	ipv4.h	PDU
IPV4Options	ipv4.h	PDU
IPV4ReqInfo	ipv4.h	
IntEmpirical	rng.h	Empirical
Interface	interface.h	Handler NotifyHandler
IpMask	node.h	
L2802_3Header	l2proto802.3.h	PDU
L2Proto	l2proto.h	Protocol
L2Proto802_3	l2proto802.3.h	L2Proto
L3Protocol	l3protocol.h	Protocol
L4Demux	l4demux.h	Protocol
L4Dummy	l4dummy.h	L4Protocol Handler
L4PDU	l4protocol.h	PDU
L4Protocol	l4protocol.h	Protocol
Link	link.h	Handler
LinkEvent	link.h	145Event
Linkp2p	linkp2p.h	Link
Location	node.h	
MACAddr	macaddr.h	

Class Name	Include File	Hierarchy
NDInfo	queue.h	
NNIfLink	qtwindow.h	
NixVectorOption	routing-nixvector.h	PDU
Node	node.h	
NodeIf	node.h	
NodeIfWeight	node.h	
NodeImpl	node-impl.h	
NodeReal	node-real.h	NodeImpl
Notification	notifier.h	
NotifyHandler	notifier.h	
Object	object.h	
OnOffApplication	application-onoff.h	Application
PDU	pdu.h	Serializable ReuseBase
Packet	packet.h	Serializable ReuseBase
Pareto	rng.h	Random
PortDemux	portdemux.h	
Protocol	protocol.h	Object
ProtocolGraph	protograph.h	
QTEvent	qtwindow.h	Event
QTWindow	qtwindow.h	QObject Handler
Queue	queue.h	
REDQueue	red.h	DropTail
RTTEstimator	rtt-estimator.h	
RTTHistory	rtt-estimator.h	
RTTMDev	rtt-estimator.h	RTTEstimator
Random	rng.h	RngStream
RandomWaypoint	mobility-random-waypoint.h	Mobility
Rate	ratetimeparse.h	
RngStream	RngStream.h	
Routing	routing.h	
RoutingEntry	routing.h	
RoutingManual	routing-manual.h	Routing
RoutingNixVector	routing-nixvector.h	Routing
RoutingStatic	routing-static.h	Routing
Seq	seq.h	
Sequential	rng.h	Random
Serializable	serializable.h	
Simulator	simulator.h	Handler
Socket	simsocket.h	
SpecificWaypoint	mobility-specific-waypoint.h	Mobility
Star	star.h	
Statistics	statistics.h	
Stats	globalstats.h	
SynFlood	application-synflood.h	Application TimerHandler
TCP	tcp.h	L4Protocol TimerHandler NotifyHandler
TCPApplication	application-tcp.h	Application
TCPDemux	tcpdemux.h	L4Demux
TCPEvent	tcp.h	TimerEvent
TCPHeader	tcp.h	L4PDU
TCPNewReno	tcp-newreno.h	TCP
TCPReceive	application-tcpreceive.h	TCPApplication
TCPReno	tcp-reno.h	TCP
TCPSend	application-tcpsend.h	TCPApplication
TCPServer	application-tcpserver.h	TCPApplication
TCP Tahoe	tcp-tahoe.h	TCP

Class Name	Include File	Hierarchy
Tfstream	tfstream.h	std::fstream
Time	ratetimeparse.h	
TimeSeq	tcp.h	
Trace	trace.h	
Tree	tree.h	
UDP	udp.h	L4Protocol NotifyHandler
UDPDemux	udpdemux.h	L4Demux
UDPHeader	udp.h	L4PDU
UDPPending	udp.h	
UDPStorm	application-udpstorm.h	Application TimerHandler
Uniform	rng.h	Random
ValueCDF	rng.h	
Waypoint	mobility-specific-waypoint.h	
WebBrowser	application-webbrowser.h	Application
WebServer	application-webserver.h	TCPApplication
WirelessGrid	wireless-grid.h	
WirelessGridManual	wireless-grid-manual.h	WirelessGrid
WirelessGridPolar	wireless-grid-polar.h	WirelessGrid
WirelessGridRectangular	wireless-grid-rectangular.h	WirelessGrid