ns-3.0.1: Pre-Alpha Design Review

ns-3 project
http://www.nsnam.org/
feedback: ns-developers@isi.edu

March 31, 2007

Introduction

This *ns-3* design document is intended to present a pre-alpha snapshot of the goals, software architecture, implementation choices, and interfaces of the *ns-3* discrete-event network simulator. It accompanies the release of a snapshot of our code.

Contents

Part I

Overview

Chapter 1

Introduction

This document accompanies the March *pre–alpha* code review release of the *ns-3* software. This snapshot is intended to provide interested readers and potential future users with a sense of the design emerging for *ns-3*, the open issues, and the near-term development plans.

The purpose of this particular release was to implement a simple simulation script (a variant of *n*s-2's `simple.tcl` script), to allow for exploration of different design proposals in the areas of node architecture, memory management, and tracing. Although a simple program is provided that yields a rudimentary packet trace, this release is by no means complete, and is not intended for use in any research. The current software consists mostly of low-level objects, is not internally consistent nor completely correct in design or interface, and does not realize the full vision the developers have for *ns-3*.

Many things in the current design are subject to change as the design team further evaluates the trade-offs between the various approaches to designing extensible, efficient, and easy–to–use network models. The tension between strong functionality, ease of use, ease of code understanding, efficiency, extensibility, and simplicity are continually being discussed and trade-offs evaluated. Feedback from the user community, and evaluation of this and successive releases, will go a long way towards helping the developers further refine the design and develop a working *ns-3*.

1.1 *ns-3* Overview

ns-3 is a discrete-event network simulator oriented towards network research and education, with a special focus on Internet-like systems. The *ns-3* project is designing a follow-on successor to the popular *ns-2* simulator.

In *ns-2*, simulation scripts are written in OTcl. In *ns-3*, simulation scripts are written in C++, with support for extensions that allow simulation scripts to be written in Python. These Python bindings have yet to be written.

ns-3 is intended to provide better support than in *ns-2* for the following items:

- Modularity of components
- Scalability of wireless simulations
- Integration/reuse of outside code
- Emulation
- Tracing and statistics

- Validation

ns-3 is a rewrite of the core of the simulator. *ns-2* does not presently run in *ns-3*, although it is for further study whether we can create a hybrid such that *ns-2* can run as part of *ns-3*, as well as which models will be ported from *ns-2* to work natively in *ns-3*.

Later chapters of this document provide more details about the software architecture, the design of nodes, packets, channels, and topologies, the supporting simulation infrastructure including random number generators, tracing, statistics, and walk-throughs of running a simple prototype *ns-3* program.

1.2 *ns-3* feature status

This section provides a rough overview of the stability of certain parts of *ns-3*, and the open issues in other areas.

Relatively stable:

- the *simulator* module: simulator, scheduler, event, and time APIs and implementation
- most of the *core* module: callbacks, asserts, debug macros, random numbers, smart pointer implementation
- these items in the *common* module: packets, buffers, and tags; data rate object
- high-level API and memory management goals

Unstable or undergoing more development:

- low-level memory management
- run-time configuration that overrides default values
- object creation (virtual constructors)
- object genericity (currently implemented via polymorphism)
- aspects of the IP node architecture and interfaces
- tracing implementation
- command-line argument parsing
- reusable topologies, frameworks, and higher-level constructs

1.3 Near-term roadmap

The development team hesitates to call this release a "pre-alpha" release because some architectural and interface aspects may be revisited in the near term. We plan to make another release in one month's time. This release is expected to be similar in functionality but with a cleaner implementation and architectural roadmap.

end-of-April:

- command-line arguments and default values
- finalize the architecture for reusable components
- finalize the memory management architecture
- finalize the tracing APIs and implementation
- stabilize key base classes and interfaces

Beyond this next release, we intend to start adding additional basic functionality:

end-of-May candidates:

- TCP/FTP
- Global IPv4 routing (static god process)
- Ethernet (simple-ethernet.cc)
- 802.11 (simple-802.11.cc) and mobility
- statistics

1.4 Longer-term vision

The PIs and developers on the project envision that *ns-3* can become more than a basic iteration of previous simulators. Here is an incomplete list of the features that are of interest to add:

- **Core refactoring:** While striving to maintain as much model reuse as possible (including a backward compatibility capability), we plan to rearchitect the simulator for better ease of use, scalability (principally by class redesign, natively supporting multi-processor and distributed simulations, and support for 64-bit machines), and support for integration of other software. The simulator should easily, with realistic models at different levels of abstraction, allow for simulations of IPv4 and IPv6 networks, as well as novel, research-oriented network architectures.
- **Software and testbed integration:** We see a tremendous opportunity, with an open-source simulator, to leverage the software developed under other open-source projects. We have three specific goals in mind:
  1. Extension of the simulation capability via integration with open-source tools and protocol implementations, including ports of popular operating system implementations;
  2. Abstraction layers, interfaces, and new techniques for supporting implementation code into the *ns-3* environment; and
  3. Techniques to allow users to easily migrate between simulation and network emulation environments.
- **Wireless models.** The *ns-2* simulator needs updating to account for the growth in wireless networking, including the many variants of IEEE 802.11 networking, emerging IEEE standards such as WiMax (802.16), and cellular data services (GPRS, CDMA). Additional new models beyond wireless are also needed, such as peer-to-peer and delay-tolerant networks.
- **Education.** *ns-3* is first and foremost a simulator for the academic research community. However, our project will emphasize making *ns-3* more useful to educators with a specific goal of its integration into undergraduate networking courses.

Chapter 2

Sample program walkthrough

To illustrate several of the concepts in the *ns-3* design, we start with a detailed explanation of a very simple *ns-3* simulation program. This will highlight several important features of the design. It is important to note that not all of the functionality described here is implemented presently, but is included to show the planned approach to topology generation, data generation, and simulation execution.

2.1 simple-serial.cc

An example, but very simple, *ns-3* main program is shown in figure 2.1-1. As mentioned in the introduction, an *ns-3* simulation is essentially a *C++* main program that instantiates the various objects that model the network being simulated and then runs the simulation to model the flow of data packets in the simulated network. The details of how to compile and execute an *ns-3* simulation are given elsewhere in this document. The purpose of this simple example is to highlight some of the important features of the *ns-3* design. The example creates a simple topology of two nodes acting as clients, two nodes acting as servers, and two routers connecting the client and server nodes. The interconnects are simple serial point–to–point links, with a fixed data rate and propagation delay. The clients have simple data generating applications and the servers have applications that receive the generated data. A step by step discussion of how to create this simulation is given in the paragraphs below.

2.1.1 The *ns-3* include files

The definitions of the various *C++* objects needed for this simulation are included using the *C++* `include` feature starting at line 6. The needed include files will of course vary depending on which *ns-3* objects are needed in the simulation.

2.1.2 Specifying the *ns-3* namespace

All of the definitions in the *ns-3* `include` files are qualified with the *namespace ns-3*. This avoids any possible name collisions with other software and libraries that might be used. Because of this, all *ns-3* simulations must either specify a default namespace as shown in line 16, or qualify all calls to any *ns-3* object or method with the qualifier `ns3::`.

2.1.3 The *C++* `main` program

Next, at line 18 is of course the definition of the *C++* main program. Not illustrated in this example are *ns-3* objects to simplify the processing of command line arguments to control any aspect of the simulation.

2.1.4 Assigning a default `Queue` object

The *ns-3* design relies heavily on the notion of *Default* values for many of the objects that might be needed. An example of this is shown at line 25. Here, the call to `Queue::Default` indicates that whenever a queue object is needed by the topology creation methods, a queue of type `DropTail` should be used. Further, the next line says that these queues should have a limit of 30 packets in the queue. Of course, it is possible to later change any given queue to a different limit, or even a different queue type. Further, there is a *default* default, that will be used if the `Queue::Default` method is not called.

2.1.5 Creating the simulated nodes

The creating of network node objects in *ns-3* is controlled by the *C++* class `Node`. The design of the `Node` objects is discussed in more detail later in this document. In this example, we want nodes that are `InternetNode`s, with a pre–configured protocol stack consisting of *IPv4*, *TCP*, and *UDP*. We could of course mix and match, and create nodes of other types. The specification of the type of node desired is controlled by a static method in the `Node` called `PushNodePrototype` shown at line 37. In this example, an `InternetNode` is pushed, which specifies that future calls to `Node::Create` will return an `InternetNode`. Of course, there is a corresponding `PopNodePrototype` which will return to the prior type of node in the prototype stack. The creation of the six node objects for this simulation are shown starting at line 39. The `Node::Create` method has responsibility for allocating a `Node` object of the correct type, and freeing the memory for this node when the simulation is complete.

Nodes can optionally have a corresponding *location*. This can be used, for example, in wireless simulations to determine whether a node can receive a transmission from another node, or in wired simulations to place the node on an animation display[1]. In this example, the nodes are assigned locations starting at line 49.

2.1.6 Connecting the nodes together

Once the nodes are created, they need to be connected together in some way. In this example, we use simple point–to–point serial links using helper methods in class `SerialTopology`. First observe that *rates* and *times* in *ns-3* are not represented by floating point values, but rather by objects of class `Rate` and `Time` respectively. These have constructors using *C++* `string` objects that specify the values in familiar notations. This can be seen at line 58.

For simple serial links, the nodes are connected using the static helper method `AddDuplexLink` method in class `SerialTopology`. This can be seen starting at line 64. This method expects two node pointers, two IP addresses, a transmission rate, and a speed of light delay. The helper actually creates two network interface devices (one for each node) with associated queues, and a serial channel object. It then configures the appropriate layer 3 protocol (*IPv4* in this example) with the address specified. It should be noted that the IP addresses specified can be any subclass of the base class `IPAddr`, which allows for configuring (for example) an *IPv6* address, or any experimental type address as long as it is a subclass of `IPAddr`.

[1] Not Implemented Yet.

2.1.7 Customizing the topology

As previously mentioned, several of the *ns-3* classes have the notion of a *Default*, which indicates what kind of object is to be used when one is created without explicit knowledge of which type is desired. An example is the creation of the queue object associated with the network interface devices in the `AddDuplexLink` described above. In line 25, this simulation specified that a `DropTail` queue with a limit of 30 packet is to be used. However, this example wants a smaller queue at the bottleneck link between the routers. The code starting at line 82 asks for a pointer to the queue between routers *r1* and *r2*, and sets the limit to a smaller value. Although not shown here, the program could have specified a completely different queue object to be used, such as a *RED* queue. Additionally, the data rates and propagation delay values between any pair of nodes can similarly be changed as needed.

2.1.8 Random Variables

An important aspect of any network simulation environment is a good set of random variables, to allow the simulation to test the performance of a network under varying conditions. At line 92, a random variable with a uniform distribution between 0 and 100 is specified, which is later used to specify a starting time for the client applications described below. Although not illustrated here, *ns-3* has a wide variety of random variables with different distributions, including uniform, exponential, pareto, weibull, sequential, empirical, and constant.

2.1.9 Adding a data demand

Once a topology has been created, the simulation must specify some type of data demand on the network, to cause the creation and forwarding of packets through the network. There are several ways to do this with *ns-3*, but the simplest one is to use one of the many `Application` classes. For readers familiar with *ns-2* or *GTNetS*, the *ns-3* applications are conceptually similar. They are associated with a single node, and make calls to the various protocols to generate data in some known or random way.

In *ns-3*, the preferred way to add an application to a node object is to use the `Add` method in the `ApplicationList` object [2]. Any subclass of the base class `Application` can be passed to the `Add` method. A new instance of the specified application is created and added to the list of applications associated with the node. Most applications will not actually start processing and generating data until the time specified by the `Start` method is reached. This insures that the simulation won't create a large amount of data all at exactly the same time. Here, the starting times for the applications are determined by sampling the `startRNG` random variable described above. In our example, the applications are added starting at line 98. In this example, the applications used are a sending application using the TCP protocol (`TCPSend`), a TCP server application (`TCPServer`), an On/Off application using the *UDP* protocol (`UDPOnOff`), and a *UDP* sink application that receives the packets generated by `UDPOnOff` (`UDPSink`).

2.1.10 Running the Simulation

Once the topology has been created and the data demand applications are added, the program can start simulating the movement of packets in the simulated network. This is shown starting at line 129. First we specify the time at which we want the simulation to stop (10 seconds in this example). Then we specify that we want a simple message printed on `stdout` at one second intervals, so we can observe that the time is actually progressing. Finally, at line 134 we enter the main event loop for *ns-3*. The `Simulator::Run` method does not exit until the specified stop time has been reached, or the event list becomes empty.

[2] It is likely that a helper function will be included to make this syntactically simpler.

2.1.11 Cleaning up the memory

Both the `Node` class and the `Simulator` class have allocated dynamic memory from the *C++* heap during topology construction and during the simulation execution. The calls to `ClearAll` starting at line 142 cause that memory to be released. Obviously, these calls are not necessary in this example, since all memory allocated by a *C++* program is automatically returned to the operating system when the program completes. However, if doing a memory leak analysis, it is important to return the memory, so that leaked memory can more easily be located.

2.1.12 Summary of simple.cc

Although quite simple, the example discussed here illustrates several important points about how to construct an *ns-3* simulation.

1. Using the *ns-3* `include` files.
2. The use of *Default* objects. This example specified that all queues be of type `DropTail` with a limit of 30 packets. Should a different queue type be desired for all queues, simply changing the default type results in all queues changing on the next simulation run.
3. Creating nodes with `Node::Create`. Using the node prototype stack and the static `Node::Create` method is the preferred way to create node object in *ns-3*. However, users can allocate their own nodes either with local variables or by allocating them from the *C++* heap. Of course, when doing this, the user retains responsibility for freeing the associated memory at a later time.
4. Setting the node location can be used for wired simulations to make the animation display show the node in a controllable way.
5. Simple topologies can be constructed easily using the `SerialTopology` helper methods. These helpers hide all of the complexity and details of connecting nodes together.
6. Applications generate data demand for the simulation, and are created using the `ApplicationList::Add` method. Users are free to use other methods to create data demands, including directly creating socket objects and/or protocols, and making the appropriate calls to the provided protocol APIs. However, for the basic and advanced users, use of the provided applications is the simplest method.
7. In this example, memory management is controlled completely by *ns-3*. Notice that there are no calls to `new` or `delete`. In general there should never be a need for an *ns-3* simulation program to call either of these. Even though the call to `Node::Create` returns a node pointer, the user is not responsible for deleting the object. Similarly the calls to `GetQueue` and `ApplicationList::Add` return pointers, and again the *ns-3* framework retains ownership and responsibility for the memory. In general, any *ns-3* method intended to be called by user program that returns a pointer retains ownership of that pointer. In short, *ns-3* programs should never *need* to allocate dynamic memory, but of course are free to if so desired.

```
1  // Demonstrate creating a simple simulation with NS3.
2  // George F. Riley, Georgia Tech, Spring 2007
3
4  // Include the various ns3 header files needed for this simulation
5
6  #include "ns3/queue.h"
7  #include "ns3/node.h"
8  #include "ns3/node-internet.h"
9  #include "ns3/process-onoff.h"
10 #include "ns3/process-tcpsend.h"
11 #include "ns3/process-tcpserver.h"
12 #include "ns3/process-udpsink.h"
13 #include "ns3/serial-topology.h"
14 #include "ns3/simulator.h"
15
16 using namespace ns3;
17
18 int main(int argc, char** argv)
19 {
20   // Optionally, specify some default values for Queue objects.
21   // For this example, we specify that we want each queue to
22   // be a DropTail queue, with a limit of 30 packets.
23   // Specify DropTail for default queue type (note. this is actually
24   // the default, but included here as an example).
25   Queue::Default(DropTail());
26   // Specify limit of 30 in units of packets.
27   Queue::Default().SetLimitPackets(30);
28
29   // The node creation in ns3 is designed to allow user specification
30   // of the "type" of node desired for each node creation.  This
31   // is done by creating a node object (the inNode below), configuring
32   // the object with the desired capabilities, and pushing the node
33   // object on the prototype stack.  In this simple example, the
34   // default behavior of an InternetNode is adequate, so we don't
35   // do any configuration here.
36   InternetNode inNode;
37   Node::PushNodePrototype(inNode);
38   // Next create the physical node topology using the node factory
39   Node* c1 = Node::Create(); // Client 1
40   Node* c2 = Node::Create(); // Client 2
41   Node* r1 = Node::Create(); // Router 1
42   Node* r2 = Node::Create(); // Router 2
43   Node* s1 = Node::Create(); // Server 1
44   Node* s2 = Node::Create(); // Server 2
45
46   // Optionally, set locations for the nodes for "pretty" animations
```

Program 2.1-1 simple.cc

```
47    // This one puts clients on left, servers on right,
48    // and routers in between.
49    c1->GetLocation()->Set(1, 1);  // Location in x/y plane
50    c2->GetLocation()->Set(1, 3);
51    r1->GetLocation()->Set(2, 2);
52    r2->GetLocation()->Set(3, 2);
53    s1->GetLocation()->Set(4, 1);
54    s2->GetLocation()->Set(4, 3);
55
56    // Define data rate and speed-of-light delays for the connecting links
57    Rate rate("10Mb");    // 10Megabits/sec
58    Time delay("10ms");   // 10 milliseconds
59
60    // Create the point-to-point links
61    // Connect clients to router r1
62    SerialTopology::AddDuplexLink(c1, Ipv4Address("192.168.1.1"),
63                                  r1, Ipv4Address("192.168.2.1"),
64                                  rate, delay);
65    SerialTopology::AddDuplexLink(c2, Ipv4Address("192.168.1.2"),
66                                  r1, Ipv4Address("192.168.2.2"),
67                                  rate, delay);
68    // Connect routers, use less bandwidth to create a bottleneck
69    SerialTopology::AddDuplexLink(r1, Ipv4Address("192.168.3.1"),
70                                  r2, Ipv4Address("192.168.3.2"),
71                                  rate/10, delay);
72    // Connect r2 to servers
73    SerialTopology::AddDuplexLink(r2, Ipv4Address("192.168.4.1"),
74                                  s1, Ipv4Address("192.168.4.2"),
75                                  rate, delay);
76    SerialTopology::AddDuplexLink(r2, Ipv4Address("192.168.4.1"),
77                                  s2, Ipv4Address("192.168.4.2"),
78                                  rate, delay);
79
80    // As an example, we reduce the queue limit on the bottleneck
81    // link to 10 packets.
82    Queue* q = SerialTopology::GetQueue(r1, r2);
83    q->SetLimitPackets(10);
84
85    // Once the topology is created, we add the processes to simulate
86    // data demand.
87
88    // Create a random variable to start the processes at random
89    // times, rather than all at time 0. In this example, we use
90    // a uniform distribution between 0 and 100 milliseconds.
91
92    Uniform startRNG(0, 100); // Random start time in milliseconds
```

Program 2.1-1 simple.cc (continued)

```
93    // Put the TCP Sending process on c1.
94    // The arguments to ApplicationTCPSend constructor are the IPAddress
95    // of the server, port number for the server, and a random variable
96    // specifying the amount of data to send.
97    Application* tcpSend = c1->GetApplicationList()->
98      Add(TCPSend(s1->GetIPAddr4(), 80, Uniform(500000, 1000000)));
99    tcpSend->Start(Milliseconds(startRNG.Value()));
100
101   // Put the TCP server on s1 and bind to port 80
102   // The argument for the constructor for the TCP server app is
103   // the port number to bind to.  Also, servers generally start
104   // at time zero, since starting a server does not generate network
105   // traffic.
106   Application* tcpServer = s1->GetApplicationList()->
107     Add(TCPServer(80));
108   tcpServer->Start(0);
109
110   // Add a UDP ON/OFF process at client c2
111   // The arguments for the ON/OFF process are the IPAddr of the
112   // destination, port number of destination, and two random variables
113   // indicating the on time and off time.
114   Application* udpOnOff = c1->GetApplicationList()->
115     Add(UDPOnOff(s2->GetIPAddr4(), 100,
116                  Exponential(Time("100ms")),
117                  Exponential(Time("100ms"))));
118   udpOnOff->Start(Milliseconds(startRNG.Value()));
119
120   // Optionally add a UDP Sink at node s2.  This not completely necessary
121   // as the packets arriving at s2 addressed to a non-bound port will
122   // simply be dropped, but the UDPSink process collects some statistics
123   // that might be useful.  The argument for UDPSink constructor is
124   // the port number to bind to.
125   Application* udpSink = s2->GetApplicationList()->
126     Add(UDPSink(100));
127   udpSink->Start(0);
128
129   // Now we are ready to start up the simulation
130   // Specify the stop time at 10 seconds
131   Simulator::StopAt(Time("10S"));
132   // Specify we want "Progress" messages at 1 second intervals
133   Simulator::Progress(Time("1S"));
134   Simulator::Run(); // Run the simulation
135   cout << "Simulation complete" << endl;
136   // At this point, we could query the udpSink object for statistics
137   // such as loss rate, jitter etc. (Not shown here).
138   // Finally, clear the memory for leak checking
```

Program 2.1-1 simple.cc (continued)

```
139     // This is only needed if we are checking for memory leaks,
140     // since (obviously) all memory allocated is returned on process
141     // exit.
142     Node::ClearAll();
143     Simulator::ClearAll();
144   }
```

Program 2.1-1 simple.cc (continued)

Although quite simple, the example discussed here illustrates several important points about how to construct an *ns-3* simulation.

1. Using the *ns-3* `include` files.
2. The use of *Default* objects. This example specified that all queues be of type `DropTail` with a limit of 30 packets. Should a different queue type be desired for all queues, simply changing the default type results in all queues changing on the next simulation run.
3. Creating nodes with `Node::Create`. Using the node prototype stack and the static `Node::Create` method is the preferred way to create node object in *ns-3*. However, users can allocate their own nodes either with local variables or by allocating them from the *C++* heap. Of course, when doing this, the user retains responsibility for freeing the associated memory at a later time.
4. Setting the node location can be used for wired simulations to make the animation display show the node in a controllable way.
5. Simple topologies can be constructed easily using the `SerialTopology` helper methods. These helpers hide all of the complexity and details of connecting nodes together.
6. Applications generate data demand for the simulation, and are created using the `ApplicationList::Add` method. Users are free to use other methods to create data demands, including directly creating socket objects and/or protocols, and making the appropriate calls to the provided protocol APIs. However, for the basic and advanced users, use of the provided applications is the simplest method.
7. In this example, memory management is controlled completely by *ns-3*. Notice that there are no calls to `new` or `delete`. In general there should never be a need for an *ns-3* simulation program to call either of these. Even though the call to `Node::Create` returns a node pointer, the user is not responsible for deleting the object. Similarly the calls to `GetQueue` and `ApplicationList::Add` return pointers, and again the *ns-3* framework retains ownership and responsibility for the memory. In general, any *ns-3* method intended to be called by user program that returns a pointer retains ownership of that pointer. In short, *ns-3* programs should never *need* to allocate dynamic memory, but of course are free to if so desired.

Chapter 3

Functional Overview

This chapter describes the *ns-3* simulator from a functional or user's perspective; i.e., without as much regard to internal implementation.

3.1 Goals

This section describes the broad goals for *ns-3*.

- *ns-3* is a discrete-event networking simulator, written in C++, with an emphasis on layers 2-4 of the OSI stack, including IPv4, IPv6, and future next-generation (non-IP) networks.
- *ns-3* is oriented towards supporting networking research and education via simulation.
- *ns-3* is free software developed using a community-oriented, open source development process, under GNU GPLv2 compatible licensing.

We want to build a system that:

1. is easy to use,
2. has replaceable components, and
3. has a base that can be used to assemble unforeseen cases from scratch

It is hard to anticipate exactly how users will use a research simulator. Therefore, we have to balance the above three goals to meet the expectations and needs of different types of users.

3.2 User experience

3.2.1 Installation

ns-3 should be buildable from source or binary formats on popular desktop and server platforms, including `x86`, `x86-64`, and `ppc`, and the Linux, OS X (Darwin), Windows (32-bit, build environment TBD), Solaris, and BSD (FreeBSD and others) operating systems.

3.2.2 User interface

- The primary *ns-3* interface is a command-line executable; it should be possible to create GUI-based configurators, but such configurators are outside the scope of the *ns-3* project.
- Simulation scripts are written as C++ main() files or as python scripts. The capability of the python script environment may be a subset of the capability of the C++ environment.
- *ns-3* should have a flexible output tracing capability that allows writing trace output to stdout, files, and other streams. Some standard tracing outputs (statistics, packet traces) will be built-in.
  - trace and log files should be convertible to the existing *nam* or out.tr format, via some internal or external scripting technique, for backward compatibility with *ns-2*.
- *ns-3* will output execution-related statistics to stdout; users may compile in other optional output to stdout.
- **Open issue: should *ns-3* be directly integrated with an animator or should the animator just run on output files (post-process)?**

3.3 Scenario definition

Users use the simulator by first *defining* a simulation scenario, *compiling* the scenario if necessary, *executing* the simulation scenario, and *processing* the simulation output, either visually through an animator, or through other handling of the output files generated. Some users will be able to run simulations by just changing command line arguments of previously defined programs. Others may require extension or recompilation of the main programs. Still others may need to edit and rebuild the core modules themselves.

- *ns-3* scenarios can be written in C++ (as a main() function), with selected configuration options exposed as command-line arguments
- *ns-3* should provide a scripting environment or interface
  - full backward compatibility with *ns-2* scripts is a non-goal
  - **Open issue: Although we have selected Python, the bindings implementation (SWIG or other) is still open.**
- scenario execution is visible on a console standard output, or written to a log file.
- support for some stock topology constructs should be provided

3.4 Tracing, logging, statistics

ns-3 plans to improve the data collection and output processing capabilities over that of *ns-2*. In *ns-2* the basic capabilities are for traces (out.tr, out.nam) and monitors. The tracing cannot be controlled at a fine granularity (e.g. trace only the foreground traffic). Monitors are primarily associated with queues.

Tracing and logging refer to simulation output that shows (with a timestamp) that a particular event occurred. Tracing generally refers to packet events, while logging refers to e.g., process log events, although the two are somewhat related.

3.4.1 Use cases

Tracing and logging are fundamental operations of the simulator. It is difficult to completely anticipate the varied future needs of the users, but here are some examples.

In the below, when we say tracing we mean both tracing and logging as defined above.

1. A typical user scenario is to build a topology and gather in a single trace file a set of ip-level queue events for every node in the topology.
2. users often want to generate multiple types of trace files: binary of ascii because trace generation is very IO intensive so they want to minimize its cost
3. users often want to enable the tracing only during certain parts of the simulation to save on trace generation which is very IO intensive and to make post-processing analysis easy.
4. users often want to enable the tracing only in certain parts of the simulation stack or only in certain nodes of the simulation topology to minimize trace generation (which is, again, very IO intensive) and to make post-processing easier.
5. users often want to perform trace analysis during the simulation: they want to calculate means or variances on certain variables and store these aggregated values rather than the full trace. Again, this is because they want to save on IO or make post-processing easier or eliminate post-processing altogether or use these statistics to change the behavior of certain simulation algorithms.

These use cases seem to lead to the following requirements:

1. decouple trace event generation from trace event serialization. (to make it possible to generate multiple trace file formats based on a single trace event data)
2. It should be possible to connect an arbitrary number of trace sinks to any trace source (for example, a routing daemon might be monitoring a bunch of trace sources while the user has setup some trace sinks on the same trace sources to generate a trace file)
3. It should be possible to disconnect a trace sink from a trace source and reconnect the same or another trace sink to the trace source later (to allow temporal-based configuration of the tracing)
4. it should be possible to connect a number of trace sources to a single shared trace sink when the trace sources are 'compatible'. That is, they generate similar data coming from different parts of the system. The interpretation of the data can be slightly different depending on where it comes from so, the sink also might need to be able to identify where the data is coming from.

What can be traced?

- Changes of value of numerical built-in POD types (ints, floats)
- Tracing (logging) of events. If we assume that events are related to function calls in the implementation, any function call should be wrappable such that input and output conditions to the function can be traced, and users should also be able to define trace points within a function call.

A goal is to minimize the need for recompilation if tracing configuration is changed. We anticipate that some but not all tracing configuration changes will require re-compilation. For instance, the tracing of a particular TCP state variable may require a TCP source code change, to change the variable from a built-in type to a Traced type.

Output trace files

Users should be able to define output file names, and to assign the output of various traces to these files. At one extreme, each trace may be written to a separate file. At the other extreme, all traces may be written to a single file (*Note:* this implies that we need to be able to identify the source of the trace element that wrote each line of the file... more on this later.).

Users should be able to declare a maximum trace file size for each trace file, and have the simulator segment the output file, so that files do not get too large. For example, if the file name "trace.out" were to be defined by the user, the simulator could automatically segment it into "trace.out.0", "trace.out.1", "trace.out.2", etc, according to some naming convention, if the file size exceeded the threshold.

Output trace format

Trace format should be selectable by the user. For instance, some users will want ASCII prints of packet trace events (like *ns-2* out.tr), others will want libpcap outputs.

The following formats have been identified.

1. text - suitable for parsing using awk, perl, etc.
2. binary - suitable for parsing using streams like in C, java, etc. and for programs like tcpdump.
3. XML - object format; slower but easier to read, convert and analyse using many tools, e.g. warehouse and statistical applications

Of course, any output trace could be further post-processed by scripts outside of the simulator.

We need to consider whether trace output files are self-describing or require metadata. For instance, one way to make them self-describing is to embed the metadata directly in the trace file:

```
-time 0.0500 -node 23 -ip_src 192.168.168.1 (etc.)
```

This has the advantage of being easy to parse but generally increases the size of the trace file substantially. An alternative might be to generate two files: "trace" and "trace.spec" with "trace" being a space-delimited set of data only

```
<SPEC_KEY_1> 0.0500 23 192.168.168.1 (etc.)
```

and a separate file "trace.spec" that has the metadata for each SPEC_KEY:

```
<SPEC_KEY_1> time node ip_src (etc.)
<SPEC_KEY_2> time interface power mac_address (etc.)
```

Regarding trace file uniformity, it is desirable to define a default uniform format for text and XML trace outputs, and to allow the user to modify this (perhaps through recompilation). Many tools that people develop to process traces depend on formats being laid out in a certain way; the desire for uniformity of format must be balanced somehow by the desire to allow users to create unconventional formats if they need to (users may need to extend or compress the standard formats).

Instrumenting a simulation for tracing

Some tracing may be built-in to the simulator and it is only a matter of the user connecting an output to capture the trace (e.g., TCP snd_cwnd may be a TracedInt type but only when the user connects a tracing collector to this object will there be any output generated.

Special commands may be defined for ease-of-use, such as "trace-all-interfaces." to get a packet trace of all interfaces in the simulator (like a tcpdump). These are assumed to be creatable in general by non-member helper functions; some will be provided by *ns-3* and users may also create their own.

Objects are generally configured for tracing when they are declared and assembled (constructed). This leads to two issues:

1. objects may be built by helper functions that do not provide low-level handles to particular components, hindering a user from configuring traces that are "non-standard" or atypical
2. related to this, objects such as nodes may be built by factories. Factories must be designed with support for tracing in mind, particularly to allow the factory to be reconfigured to generate a node with a different trace configuration than the previous.

Statistics

users often want to perform trace analysis during the simulation: they want to calculate means or variances on certain variables and store these aggregated values rather than the full trace. Again, this is because they want to save on IO or make post-processing easier or eliminate post-processing altogether or use these statistics to change the behavior of certain simulation algorithms.

Special tracing requirements and open issues

- Ability to generate pcap formatted tracefiles, for use in analyzers such as Ethereal or the many utilities designed to interpret libpcap output files.
- Ability to take (parse) pcap files as input to traffic generators (under discussion on ns mailing list).
- Some facility to uniquely identify all trace sources in the trace outputs, if needed. For libpcap output, this type of output is not needed, but for "out.tr"-style formats where multiple trace sources write into the same file, some means of uniquely identifying the name of the object that generated a line of trace is needed.
- A useful feature for me has been the ability to define start/stop points in time to enable/disable all tracing (e.g., start tracing at time X, stop at time Y). It would be nice if there were a single-line command to enable this globally.

- For non-pcap packet traces, the use of a unique packet sequence number has historically been very useful. Uniqueness across a simulation environment is a challenge if the simulation is distributed, or if there is emulation involved, so this goal may only be met by single sequential simulations not involving emulation.

3.5 Miscellaneous capabilities

- *ns-3* shall provide an emulation capability– ability to source/sink real packets and execute in real-time
- *ns-3* shall be designed to scale for parallel processor support and distributed simulations, in a manner mostly transparent to users.
- *ns-3* shall provide interfaces that facilitate the porting of implementation code (user space and kernel TCP/IP stacks)

3.6 Documentation

- Source code APIs shall be documented using Doxygen and available on the web as HTML and Latex-generated PDF.
- the various project documents use Latex.

The documentation should be checked out nightly, built into PDF and HTML, and posted on the web. Documentation should be stored in the source code repository in source form, with figures stored in eps or image format. Vector graphics should be drawn in a commonly available vector graphics program such as dia, tgif, or xfig, and the sources stored with the eps.

Chapter 4

Use cases

This chapter identifies the types of use cases that we envision for *ns-3*. This includes both a definition of the classes of users that we anticipate, and the types of things they will intend to do with the simulator.

The *ns-3* software package is a set of *C++* class definitions and implementations that allows users to construct simulations of computer networks. It includes models for communication links, network interface devices, network nodes, protocols, data packets, user applications, and much more. Once the *ns-3* code has been compiled and the libraries built, creating a working simulation can be as simple as creating a *C++* main program, instantiating one of the many built-in topology objects, adding one or more user applications to the network nodes, and running the simulation. A more advanced user can create a new protocol at any layer, insert that new protocol into the protocol stack at one or more nodes, and observe the behavior of the new protocol. Alternately, he can easily study the effects of modifications to existing protocols by using the *C++ sub–classing* feature and implementing new versions of one or more of the virtual functions. For example, the base class *TCP* has virtual functions for managing duplicate *ACK* packets, timeouts, and new data acknowledgements[1]. Thus a new congestion control algorithm can be designed and evaluated, without any modifications at all to the basic *TCP* model. The most advanced users can extend or enhance the network models found in the *ns-3* code base, or design and contribute new models to suit their needs.

We have identified three basic classes of users for the *ns-3* simulator.

1. The *Basic* user will primarily use the existing *ns-3* models as implemented by the development team, and construct simulations using those models connected together in simple to moderately complex ways. This class of user will primarily focus on using the pre–defined network nodes (for example an `InternetNode` configured with an *IPv4*, *TCP*, and *UDP* protocol stack), and using the built–in applications (for example a web browser and web server model) to generate data demand for the network. This user will likely compile the basic *ns-3* libraries only once, and simply compile their *C++* programs to link against the *ns-3* libraries. The *Basic* user does not need to be a *C++* expert, but does need a basic understanding of the syntax of *C++* programs. He does not need to understand class hierarchies or sub–classing.

2. The *Advanced* user will extend *ns-3* by adding new protocols (often by subclassing existing protocols, but not necessarily), or modifying the behavior of existing protocols in some way. The flexible design of *ns-3* allows this to often be accomplished without modification to the *ns-3* models, using virtual functions from existing protocols, or by inserting new protocols into a protocol stack. Again, it may be that this class of user rarely re–compiles the *ns-3* libraries, but in some cases might need to. The *Advanced* user needs a strong understanding of *C++* programming concepts, including subclassing, virtual functions, static functions, and polymorphism.

3. The *Expert* user will develop new models for *ns-3*. These might be new routing protocols, new wireless MAC protocols,

[1] Not Implemented Yet.

new transport protocols such as SCTP, or overlay network protocols. This user will likely (but not necessarily) need to modify or extend *ns-3* modules, re–compile the *ns-3* code base, and create main programs that instantiate the new models. The *Expert* user needs all of the skills of the advanced user, plus familiarity with *C++* templates, memory management, and complex code–build systems.

In the sections that follow, they will often be annotated with the class of user that will likely be concerned with the material in those sections. A basic user can safely skip the material in sections marked by *Advanced* or *Expert*. Sections with no annotations should be read by all users.

4.1 Use of Example Programs

To Be Completed

4.2 Modification of Example Program

To Be Completed

4.3 Topology Creation

To Be Completed

4.4 Distributed Simulation

To Be Completed

4.5 Network Emulation

In this use case, we show how *ns-3* can be used to send real packets over real networks instead of over simulated links.

To Be Completed

4.6 Research Use

The primary goal of the *ns-3* project is to produce a high quality research simulator. Here we illustrate how an internet protocol suite can be imported, modified and analyzed using the features of *ns-3*. This use-case illustrates how outside software can be imported into *ns-3* and analyzed using the statistics and tracing mechanisms we provide.

4.6.1 Background

It is expected that there will be several relatively broad classes of research applications.

4.6.2 Simulation

For the first class, the maximum number of nodes is typically 200. For the second class, very large-scale p2p applications, we should support simulations in the 5,000 node range where current simulators become limited by memory usage.

For users in the 200-node arena, we have determined that many of these users don't really care about access to layer interface objects. They just want to set up nodes and connect them using links. Most users of this kind of simulation don't care if IP addresses are assigned to interfaces or nodes. They don'really care if nodes are identified by IP address at all. They want to be able to do something like the following pseudocode:

```
Node *a = new Node ();
Node *b = new Node ();
WirelessMedium *medium = new WirelessMedium ();
medium->Connect (a);
medium->Connect (b);
Source *source = new Source (a);
source->SetDestination (uid);
```

The "uid" is a value which represents the abstract address of a node. Nodes get get implicitely-assigned addresses based on an increasing global counter.

Another class of user wants to more faithfully use the IP system. They have a need for some kind of IP-level automatic routing. These users want to usee real IP but want to avoid having to manually configure IP addresses and forwarding tables.

An even smaller class of users need to change the routing tables. This argues for a routing subsystem (module) with a simple API available such as:

```
StaticIpRouter::ConfigureForwardingTables (void).
```

More elaborate systems which would avoid the cost of maintaining in-memory forwarding tables with O(n) entries are nice but not strictly necessary for the roughly 200 node topologies which are typical of their use-cases.

NIx-Vector routing would also be nice but is independent on both static configuration and forwarding table size optimizations. At the highest level, enabling NIx-Vector routing should also be as simple as calling an Ipv4::Enable () method prior to running the simulation. This perhaps calls for a derived requirement for componentization of routing and address resolution functionality.

4.6.3 Emulation

There is a class of users who work on emulation and use tools such as PlanetLab. For them, the use-case is very different: They want to write real applications which run on real test beds and they want to run those same real applications in a simulator before deploying their applications to hundreds or perhaps thousands of sites. They want to debug the applications and evaluate their behavior in an easy way prior to committing to real hardware.

For this class of users, the fact that IP addresses are assigned to interfaces rather than nodes does matter because this is what the test-bed and evnetually the real world will be doing. These users also typically want a blocking socket API very much like the posix socket API. For them, the task of topology construction and configuration is really decoupled from the actual network application code. The key to satisfying this class of users is to allow them to create and configure an IP topology with wired and wireless links and then to allow them to instantiate their application and make it run on top of this topology. They want to see something that allows them to run an application as if it were a new process beginning user-leel execution at main ().

```
  package myApplication
  int my_main_function (int argc, char *argv[]) {
    // behave like an application in an address space
    // create tcp socket, call connect to establish connection,
    // send and receive data.
    // do application-like things
    // and finally exit
    return 0;
  }
//
// Create a process / thread that begins executing the ‘‘applciation.’’
//
  Process *process = new Process (&my_main_function);
```

4.6.4 Tracing

In the research use-case, tracing is very important but very diverse in requirements and implementation.

In a broad sense, some researchers do all their analysis within the simulation itself and output aggregated results. Others generate gigabyte-size traces and run complex post-processing steps. Some others generate large traces and use these to display nice-looking animations (for a wireless mobile network, for example, you can display the moving nodes by a dot at a pixel position and the corresponding queue occupancy with a small vertical stick).

Most people using tracing actually modify the core code to introduce trace sources where they need them; typically by introducing a trace hook by adding a call to a global function, potentially with some context-specific arguments.

4.7 Code reuse

We would like to see an architecture that strongly supports code reuse at both the kernel and application level.

1. experience shows that it is really long and tedious to have to reimplement already implemented protocols for a simulator, or reimplement them when moving from simulation to implementation
2. people want to be able to conduct simulations that really faithfully mirror the implementation code
3. we need to reuse *ns-2* and other code

For kernel-level reuse, we would really like to see a Linux TCP/IP node that ports the Linux IPv4/IPv6/TCP/UDP, etc. stack. The network simulation cradle (http://research.wand.net.nz/software/nsc.php) is probably the best example here. We also think that APIs that mimic the Linux or BSD networking APIs (user-to-kernel APIs such as sockets, pfkey, netlink, etc.)

and also configuration (proc filesystem) should be the initial template for how to design our interfaces; not because they are perfect, but because they are familiar.

For application level code, we would like to see an environment built-in to *ns-3* to allow application writers to write code that can run as an *ns-3* object but also as a user-space process. There have been several public and private libraries built to do this sort of thing for OPNET, ns-2, QualNet, etc. One that is of particular interest is the protolib toolkit at NRL, which has been heavily used: http://cs.itd.nrl.navy.mil/work/protolib/index.php

Regarding *ns-2* code reuse, it would be nice if we could port *ns-2* transport agents, and other modules, to *ns-3*. Moving the TCP code over should be a high priority item.

Another way to reuse code is to be able to reuse tools that process network traces or provide network trace data. Pcap is one of the best examples here, but other examples (snort files, route views, etc.) are also relevant.

4.8 Emulation

Emulation is somewhat related to the topic above, because users want to reuse simulation code also to run in testbeds and for experiments over the network.

The trend in networking research is to allow testbeds and emulators to play a more significant role in research. Some steps have been taken with Emulab and Planetlab but more could be done. We would like an architecture that facilitates integration with testbeds (ORBIT, Emulab, and PlanetLab in particular). This probably means working more closely with these communities in the initial stages of our development, rather than grafting on later.

4.9 Heterogeneity

Although our current focus is on basic IPv4 network scripts, we should keep in mind that we want users to use our tool for simulating IPv6, IPv8, and stranger network types like disruption tolerant networks (DTNs), underwater acoustic networks, sensor networks, and some of the redesigned stacks for e.g. DARPA wireless programs and probably eventually NSF GENI/FIND programs.

What this means for our simulator architecture is that we will need to think of what are some of the more general APIs vs. some of the more IPv4-specific APIs and try to ensure that we do not couple too much Ipv4 into APIs when not needed. For example, a Topology object providing a random-waypoint mobility model should not have any IPv4 required dependency, although perhaps it is overloaded to have some IPv4 relationship.

4.10 Scalable tracing

Tracing needs can be quite varied, even for an individual user. Sometimes, users need to parse the ns-2 "out.tr" tracefiles with post-processing scripts. Other times, simulations have required very intensive modifications of the code of particular modules to dump out detailed event logging (in a routing daemon, for example). Large simulations may not have per-packet tracing enabled, and only selected statistics being outputted at the end.

While pcap tracing is important for *ns-3*, to allow tcpdump-related utilities to work, many users will not be able to use those "built-in" traces and will need flexible statistics gathering and event logging capabilities.

4.11 Educational Use

One of the goals of the *ns-3* project is to produce a simulator that can be used for educational purposes. Here we show that *ns-3* can be used in undergraduate networking courses.

A possible priority here is to provide a set of easily clonable and modifiable scripts that can be used in an educational context. For example, simulation scripts that do all of the exercises in a popular text. This is important because students often learn by cloning and tweaking past working code rather than starting from scratch.

A good first goal here may be to port all of the samples in Calvert and Donahoo's book to *ns-3*:
http://books.elsevier.com/mk/default.asp?ISBN=1-55860-826-5

Chapter 5

Architecture

This chapter provides an introductory software architectural overview of *ns–3* including source code organization, memory management, and (others TBD).

5.1 Basics

ns-3 is a user-space program that runs on Unix- and Linux-based systems and on Windows (build process, via Cygwin or via native win32 APIs, is to be determined). It contains support for the following:

- construction of virtual networks (nodes, channels, applications) and support for items such as event schedulers, topology generators, timers, and the like to support discrete-event network simulation focused on Internet-based and possibly other packet network systems.
- support for network emulation; the ability for simulator processes to emit and consume real network packets
- distributed simulation support; the ability for simulations to be distributed across multiple processors or machines
- support for animation of network simulations
- support for tracing, logging, and computing statistics on the simulation output

ns-3 is written in C++, with a planned Python scripting interface(s) for users.

ns-3 has a modular implementation containing a core library supporting generic aspects of the simulator (scheduler, events, packets, random number generators, tracing, logging, and statistics, etc.) and a few abstract base classes, just to get the architecture and interfaces defined consistently.

5.2 Source code organization

Figure 5.1 provides an overview of the *ns-3* source code organization. Section A.2 below details the build environment and options. The ns-3 library is split across multiple modules:

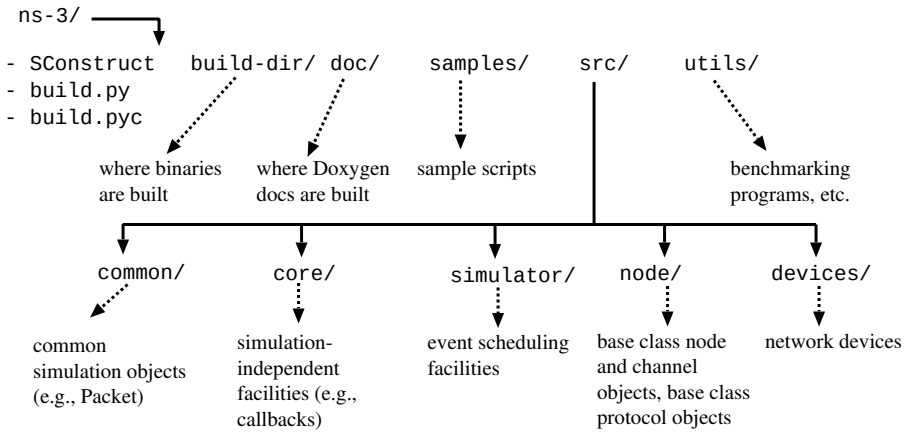

Figure 5.1: Current code organization for *ns-3* project.

- core: located in src/core and contains a number of facilities which do not depend on any other module. Some of these facilities are OS-dependent.
- simulator: located in src/simulator and contains event scheduling facilities.
- common: located in src/common and contains facilities specific to network simulations but shared by pretty much every model of a network component.

A number of files exist in the top-level directory (SConstruct, build.py, and build.pyc) to coordinate the build process. The details of the build process are found in Appendix A.

5.3 Memory Management

Memory management in a *C++* program is a complex process, and is often done incorrectly. Memory can is allocated globally, using variables at global scope or static member variables in classes, locally on the stack using local variables with limited lifetime, or dynamically using memory allocation methods such as `new` or `malloc`. In the third case, any memory allocated dynamically must always be returned[1].

Our design philosophy is based in part on the assertion that a user should never have to allocate dynamic memory. When making calls to the *ns-3* API that require memory allocation, the *called* function will allocate the memory as needed, and retain responsibility for deallocating it. An example is when creating node object, such as:

```
Node* n1 = Node::Create();
```

The memory for the node object is allocated by the `Create` method, and a pointer to the new object is returned. The responsibility for deleting the memory is retained by the `Node` class. This is referred to as retaining *ownership* of the memory.

A slightly more complicated example is in the creation of new application objects. There are many different types of applications, and it would not make sense to have separate API calls to create each one. Since all applictions must derive from the base class `Application`, we have a single API to create any application that derives from this base class:

[1]This is not precisely true. Memory allocated with a lifetime equal to the lifetime of the owning process is automatically deallocated when the process completes, and thus technically does not need to be explicitly deallocated. For *ns-3*, we are specifically deallocating memory even in this case, for completeness and for ease of memory leak tracking.

```
MyNewApp* myApp = appManager->Add(MyNewApp());
```

The memory for the new application is allocated by the `Add` method by copying (using the required `Copy` method) the application passed as an argument. Here, the argument is an anonymous, temporary object of class `MyNewApp`. Again, the ownership of the memory is retained by the application manager, and the user in is not responsible for deallocation.

The basic design principle used by *ns-3* is "You create, you delete". Here, "*You*" refers to the class allocating memory. "*Create*" refers to allocating dynamic memory either with `new` or by calling the `Copy` method on an existing object. "*Delete*" refers to freeing the associated memory, either when the owning object is destroyed, or when explicitly asked to by the object itself.

A variation on this principle which is also acceptable within *ns-3* is "You create, you insure deletion". Here, "*insure deletion*" refers to assigning ownership of the memory to some other object, using a well–defined and well–documented interface.

Users using a low-level API who wish to explicitly allocate objects on the heap, using operator new, are responsible for deleting such objects.

Note: The description above of memory management is not completely implemented in the ns-3.0.1 release.

Part II

Core

Chapter 6

ns-3 core

This chapter discusses the design and implementation of core elements in *ns-3*. These items are built in two modules (`core` and `simulator`) with no other dependencies on the simulation code. Figure 6.1 illustrates the portion of the code described in this chapter.

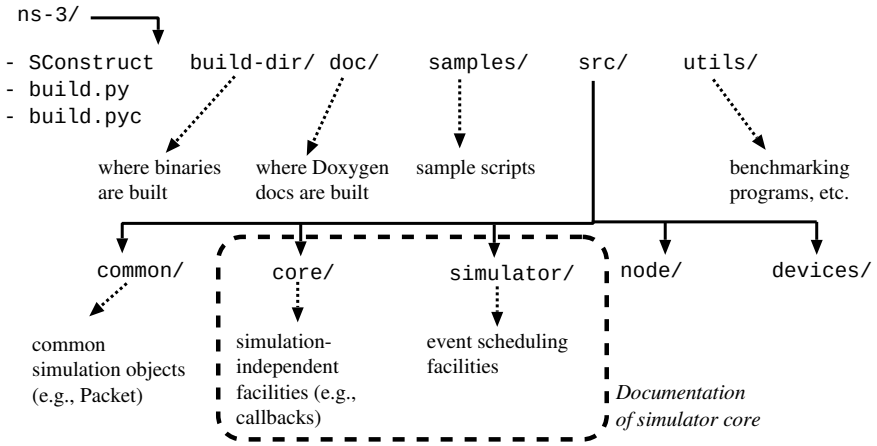

Figure 6.1: Source code (within dashed oval) described in this chapter.

The items described in this chapter include:

- classes Simulator, Scheduler, and Event (and related classes)
- representation of simulation time
- facility for defining callbacks
- reference list implementation
- system-dependent handling of file I/O and system time

6.1 Simulator, Scheduler, and Events

This section is concerned with the general structure of the simulator for coordinating the execution of simulation events.

- a base Simulator class
- an Event class and some facility for handling events
- a base Scheduler class designed to hold Events
- some facility for callbacks
- support for timers

6.1.1 Simulator

At its most basic, a Simulator object provides a public interface that allows objects to insert Events into a Scheduler, and keeps track of simulation time elapsed.

The Simulator class provides a single static Simulator object. The most common operations are to call `schedule()` to add events to the scheduler. The below program snippet gives an example:

```
int main (int argc, char *argv[])
{
    MyModel model;

    Simulator::schedule (Time::absS (10.0), &random_function, &model);

    Simulator::run ();

    Simulator::destroy ();
}
```

6.1.2 Scheduler

The Scheduler is an object that dynamically stores Events in some type of ordered data structure. There are various structures possible (linked list, heap, map, etc.) to hold the events; depending on the type and number of Event manipulations required in a simulation, one type of underlying scheduler may perform better than another.

The simulator supports runtime replacement of the underlying event scheduler through the base class `Scheduler`.

The three provided schedulers are:

- Linked List (insert: O(n), remove: O(1)
- Binary Heap (insert: O(log(n)), remove: O(log(n)))
- Std Map (insert: O(log(n)), remove: O(log(n)))

plus the simulator allows a user to insert his or her own scheduler.

The scheduling order of events scheduled to expire at the same time is specified to be that of the insertion time. i.e.: the events inserted first are scheduled first. This order holds whatever the scheduling algorithm chosen: it is implemented by using an event sequence number, incremented for each insert.

6.1.3 Events

An Event is an object that tells the simulator to do something at a specific time. There are a few related concepts to discuss here: events, timers, and callbacks. We start by describing Mathieu Lacage's yans simulator; the basis for the *ns-3* event design..

The yans event design: A yans event is a wrapper around a C++ method or function. A yans Event is a wrapper around a C++ method or function that also holds the value of the arguments passed to the method/function. The method consuming the event can be any method in scope (possibly with arguments). The Event subclass is simple, providing a pure virtual `notify()` that consumes the event (you need to invoke delete on an Event explicitly after calling notify).

The yans Event is similar to the *ns-2* Handle; the departure from *ns-2* way of doing things is that a set of templates are used to automatically generate the code for the subclasses of the Event/Handler class. These automatically-generated classes are 'forwarder' classes which 'forward' the event notification to arbitrary functions or class methods with an arbitrary number of arguments. These templates implement a version of the Command design pattern. In terms of Boost, a yans Event is a fully-bound functor.

These templates export a single overloaded function to the user: `make_event (method, arguments)` which is a constructs the right type of Event from a method or function pointer and the arguments. The Event object is then passed down to the simulator's schedule methods.

The *ns-3* event design: *ns-3* has adopted the technique used in yans with a small change: the `make_event` method has been integrated in the in the main `Simulator::schedule` method which saves users quite a bit of typing. *ns-3* thus does:

```
Simulator::schedule (time, &method, arguments);
```

while yans does:

```
Simulator::schedule (time, make_event (&method, arguments));
```

The *ns-3* version also makes the memory management of events simpler and allows us to avoid using smart pointer and refcounting to manage the memory associated to events.

Cancelling an event There are two basic options to cancel an event:

- set the cancel bit on an event: when the event expires, if its cancel bit is set, we do not run the event's notify method. This operation usually has O(1) algorithmic complexity
- remove the event from the event list. This operation usually has at least O(log(n)) complexity.

Although both methods have the same semantics, they have different complexity behaviors. Currently, *ns-3* supports both through the following methods:

- `Simulator::cancel (eventId`: set the cancel bit
- `Simulator::remove (eventId)`: remove from event list

For convenience, the EventId class also exports a cancel method.

6.2 Timers

In *ns-3*2, timers (class TimerHandler) are derived from Handlers. At a high-level, they wrap Events that are inserted into the scheduler, providing methods like cancel(), sched() and resched().

In *ns-3*, timers are simply Events; the method to call upon expiry is stored in the Event as a function pointer, and the Event may be cancelled by either explicit removal of the Event from the scheduler queue, or by setting a cancel bit.

6.3 Time

Simulation time may either be represented as a floating point or integer number internally. Historically, the handling of events in *ns-2* on different platforms led to different event ordering due to floating point arithmetic differences (rounding) on different platforms. In *ns-3*, time is maintained internally as nanosecond integers (stored internally in a `uint_64`).

ns-3 provides a Time class for safer usage of simulation time. The class provides a number of static member functions allowing users to create a Time object representing time according to different units; examples include:

- `Time::absS(double s)` Return Time object corresponding to an absolute time of s seconds into the simulation.
- `Time::relS(double s)` Return Time object corresponding to a relative time of s seconds beyond the current simulation time.
- `Time::now(void)` Return Time object corresponding to the current simulation time.

ns-3 also provides a TimeUnit template class for keeping track of units. This template class is used to keep track of the value of a specific time unit: the type TimeUnit<1> is used to keep track of seconds, the type TimeUnit<2> is used to keep track of seconds squared, the type TimeUnit<-1> is used to keep track of 1/seconds, etc.

This base class defines all the functionality shared by all these time unit objects: it defines all the classic arithmetic operators +, -, *, /, and all the classic comparison operators: ==, !=, <, >, <=, >=. It is thus easy to add, substract, or multiply multiple TimeUnit objects. The return type of any such arithmetic expression is always a TimeUnit object.

The ns3::Scalar, ns3::Time, ns3::TimeSquare, and ns3::TimeInvert classes are aliases for the TimeUnit<0>, TimeUnit<1>, TimeUnit<2> and TimeUnit<-1> types respectively.

For example:

```
Time<1> t1 = Seconds (10.0);
```

```
Time<1> t2 = Seconds (10.0);
Time<2> t3 = t1 * t2;
Time<0> t4 = t1 / t2;
Time<3> t5 = t3 * t1;
Time<-2> t6 = t1 / t5;
TimeSquare t7 = t3;
Scalar s = t4;
```

If you try to assign the result of an expression which does not match the type of the variable it is assigned to, you will get a compiler error. For example, the following will not compile:

```
Time<1> = Seconds (10.0) * Seconds (1.5);
```

Building on this, the class Time is an instance of ns3::TimeUnit<1>:

```
class Time : public TimeUnit<1>
```

This Time class is used by the return value of the ns3::Simulator::Now method and is needed for the Simulator::Schedule methods

Time instances can be created through any of the following classes:

- ns3::Seconds
- ns3::MilliSeconds
- ns3::MicroSeconds
- ns3::NanoSeconds
- ns3::Now

Time instances can be added, substracted, multipled and divided using the standard C++ operators (if you make sure to obey the rules of the ns3::TimeUnit class template) To scale a Time instance, you can multiply it with an instance of the ns3::Scalar class. Time instances can also be manipulated through the following non-member functions:

- ns3-Time-Abs ns3::Abs
- ns3-Time-Max ns3::Max
- ns3-Time-Min ns3::Min

6.4 Callbacks

The callback API in *ns-3* is designed to minimize the overall coupling between various pieces of of the simulator by making each module depend on the callback API itself rather than depend on other modules. It acts as a sort of third-party to which work is delegated and which forwards this work to the proper target module. This callback API, being based on C++ templates, is type-safe; that is, it performs static type checks to enforce proper signature compatibility between callers and callees. The API is minimal, providing only two services:

- callback type declaration: a way to declare a type of callback with a given signature, and,
- callback instantiation: a way to instantiate a template-generated forwarding callback which can forward any calls to another C++ class member method or C++ function.

The implementation is based on use of templates to implement the Functor Design Pattern. It is used to declare the type of a Callback:

- the first non-optional template argument represents the return type of the callback.
- the second optional template argument represents the type of the first argument to the callback.
- the third optional template argument represents the type of the second argument to the callback.
- the fourth optional template argument represents the type of the third argument to the callback.
- the fifth optional template argument represents the type of the fourth argument to the callback.
- the sixth optional template argument represents the type of the fifth argument to the callback.

Callback instances are built with the makeCallback template functions. Callback instances have plain old data (POD) semantics: the memory they allocate is managed automatically, without user intervention which allows one to pass around Callback instances by value.

Walk-through the below example

```
/* -*-    Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:f -*- */
#include "ns3/callback.h"
#include <cassert>
#include <iostream>

using namespace ns3;

static double
cbOne (double a, double b)
{
    std::cout << "invoke cbOne a=" << a << ", b=" << b << std::endl;
    return a;
}

class MyCb {
public:
    int cbTwo (double a) {
        std::cout << "invoke cbTwo a=" << a << std::endl;
        return -5;
    }
};


int main (int argc, char *argv[])
{
    // return type: double
    // first arg type: double
    // second arg type: double
```

```
    Callback<double, double, double> one;
    // build callback instance which points to cbOne function
    one = makeCallback (&cbOne);
    // this is not a null callback
    assert (!one.isNull ());
    // invoke cbOne function through callback instance
    double retOne;
    retOne = one (10.0, 20.0);

    // return type: int
    // first arg type: double
    Callback<int, double> two;
    MyCb cb;
    // build callback instance which points to MyCb::cbTwo
    two = makeCallback (&MyCb::cbTwo, &cb);
    // this is not a null callback
    assert (!two.isNull ());
    // invoke MyCb::cbTwo through callback instance
    int retTwo;
    retTwo = two (10.0);

    two = makeNullCallback<int, double> ();
    // invoking a null callback is just like
    // invoking a null function pointer:
    // it will crash.
    //int retTwoNull = two (20.0);
    assert (two.isNull ());

    return 0;
}
```

6.5 File I/O, system time, and reference list implementation

The "core" module (`ns3/src/core`) contains additional cross-platform utilities:

- `class SystemFile`: an OS-independent interface to get write-only access to a file
- `class SystemWallClockMs`: an OS-independent interface to get access to the elapsed wall clock time
- `class ReferenceList`: a templatized implementation of a reference list, based on the description of the technique found in "Modern C++ Design" by Alexandrescu (Chapter 7).

Chapter 7

Running ns-3 simulations

The previous chapter detailed some core implementations of various aspects of the simulator. This chapter builds on the previous by describing the elements of writing rudimentary programs to use the core. Subsequent chapters detail the design and implementation of other features (packets, channels, nodes, etc.) necessary to run useful network simulations.

This chapter walks through a simple example script, found in the location `samples/main-simulator.cc`.

7.1 Executing simulations

In *ns-3*2, simulation scripts are written in OTcl. In *ns-3*, simulation scripts are written in C++, with support for extensions that allow simulation scripts to be written in Python scripting language. These Python bindings have yet to be written. A more detailed walkthrough of a more usable program can be found in Section 2.

7.1.1 main() program

Simulations are executed as C++ programs that link compiled object code and libraries, instantiate and create bindings between objects, and create a simulator object that controls the running of the objects in simulation time.

A basic skeleton of what this will look like is below:

```
/* -*-    Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:f -*- */
#include "ns3/simulator.h"
#include "ns3/time.h"
#include <iostream>

using namespace ns3;
```

The first few lines include an emacs mode line, include files, and a using directive to pull in the `ns3` namespace. This line is important because all core *ns-3* code is found in the `ns3` namespace. Users can define and include other namespaces as well.

Next, let's create a simple dummy class. This class has two member functions; a `start()` function that schedules an event

at ten seconds after it is invoked, and a function to deal with the event, using the event scheduling technique described in the previous chapter.

```
class MyModel {
public:
    void start (void);
private:
    void dealWithEvent (double eventValue);
};

void
MyModel::start (void)
{
    Simulator::schedule (Time::relS (10.0),
                &MyModel::dealWithEvent,
                this, Simulator::now ().s ());
}
void
MyModel::dealWithEvent (double value)
{
    std::cout << "Member method received event at " << Simulator::now ().s () <<

        "s started at " << value << "s" << std::endl;
}
```

Now, for good measure, let's show how one schedules a random function to execute at some time in the future. This function, upon being called, will start the MyModel object.

```
static void
random_function (MyModel *model)
{
    std::cout << "random function received event at " <<
        Simulator::now ().s () << "s" << std::endl;
    model->start ();
}
```

Finally, the `main()` program is pretty simple.

```
int main (int argc, char *argv[])
{
    MyModel model;

    Simulator::schedule (Time::absS (10.0), &random_function, &model);

    Simulator::run ();

    Simulator::destroy ();
}
```

and it produces the following output:

```
random function received event at 10s
Member method received event at 20s started at 10s
```

In the next chapters, we document how more useful network simulation objects are designed and implemented, before later walking through a more sophisticated simulation program example.

7.1.2 Scripting interface

Open issue: How to support scripting.

It seems generally accepted that *ns-3* should have a scripting interface, not based on OTcl, but opinions differ on what is the right alternative. There have been a few options suggested:

- SWIG (http://www.swig.org)
- boost.python
- Ruby
- reuse/generalize the GIMP PDB (http://www.advogato.org/article/550.html).

It may be likely that we get some novel contributions from the community on how to do scripting in multiple ways.

Recent discussions with the M5 simulation project suggest that Python is a good choice, with possibly support for SWIG bindings. Mathieu Lacage plans to add some hooks for Python scripting.

Part III

Nodes, Packets, Channels

Chapter 8

Node Architecture

8.1 Overall node architecture

Nodes are simulation objects that represent physical devices on networks. These can be end systems (desktops of laptops), routers, hubs, wireless handheld devices, or any device with one or more network interface devices for connecting to a network. In our design, a Node object can be thought of as container objects for individual elements (interfaces, protocols, applications) that are bound together to form a networked computing device.

There are several design decisions that must be made regarding the Node object, specifically:

1. How are nodes to be created?
2. What class hierarchy (if any) is to be used.
3. What is the API to access the various objects within a node.

8.1.1 Node construction and deletion

The design of the node construction and deletion framework was driven by the following requirements:

1. Every node should have a globally unique numeric identifier, assigned automatically by the ns-3 node creation framework.
2. There must be an easily accessible list of every node in the toplogy, to allow users to obtain a pointer to every node in turn, and configure or otherwise manipulate the node contents and/or behavior.
3. The memory allocated for node objects should be deleted, either automatically or by a user API call.
4. Users should be able to specify any particular node object and node configuration to be used for subsequent node creation requests. This is useful when creating *Stock Topology Objects*, such as a dumb-bell or start topology. The stock topology object will simply ask for a node object and get a node base class pointer. The actual type of node will be as previously specified by the user.
5. Users should be able to create node ojects on the stack using local variables, or on the heap using `new`. In the latter case, the user retains responsibility for deleting the memory at the end of the simulation.

To achieve these goals, we have designed a node creation framework using static methods in the node class. The framework is based on the notion of a *node prototype stack*. The API has methods to add a node object (any subclass of a node and configured as desired by the user) to the top of the stack, and of course popping the stack. Users then call the static `Create` method in the node object to obtain a pointer to a node object as defined by the top of the stack. A second `Create` method is defined to allow creation of a specific node subclass, with a typesafe return, as defined below.

1. The preferred method for creating a node object is using one of the static `Node::Create` methods as follows:
   (a) `Node* mynode = Node::Create()`. This returns a pointer to a node object of the same subclass as the top of the stack, and configured identically as the top of the stack.
   (b) `MyNodeSubclass* mynode = Node::Create(MyNodeSubclass())` This creates a new node object of class `MyNodeSubclass` and returns a pointer to the node object in a typesafe manner.

   In both of the above cases, the ownership and responsibility for deletion are retained by the Node class. Users do not delete the node objects individually; rather they are deleted by a call to `Node::ClearAll()` when the simulation has completed. Further, in both cases, a globally unique identifier is assigned to the nodes, and the node pointer is retained in a global list of all nodes in the simulation.
2. A secondary, but less desirable, way to create nodes is by creating local variables or allocating nodes from the heap. In either of these two cases, the user must explicitly add the node object to the global list of known nodes, which also assigns a unique identifier. For example:
   (a) `MyNodeSubclass* mynode = new MyNodeSubclass(); Node::Add(mynode);`
   (b) `MyNodeSubclass mynode; Node::Add(&mynode);`

   In both of the above cases, the node is assigned a unique identifier and added to the global list of nodes in the simulation. The ownership of the pointer is *NOT* transfered, and the responsibility for deleting the `new'd` pointer is retained by the user.

8.1.2 Node Capabilities

Designing an extensible class hierarchy for the node objects was considered and rejected by the design team. The team believes there is no way to anticipate all possible combinations of node features and options such that a class hierarchy will be useful. Rather, the node class hierarchy is designed to be a simple *tree* hierarchy, with a single base class `Node`, and then individual node objects deriving directly from `Node`.

The flexibility to allow any node subclass access to any defined behavior is achieved by the design of *Node Capabilities*. The design approach is to create a node object by including one or mode "node capabilities", selecting the capabilities based on the desired features and behavior of a node. For example, an "InternetNode" has capabilities for a list of network devices, a layer 3 protocol list, a layer 4 protocol list, and a list of applications. A "SensorNode" has a list of network devices, a list of "Sensors", and an energy model. The base class defines none of the capabilities; rather subclasses define which of the capabilities are appropriate for the the subclass. However, the base class does define a virtual `Get` method for every known capability. Thus an owner of a base class node pointer can get a pointer to any capability (or a null pointer if the specific cabability is not appropriate for the specific subclass). The capabilities can themselves be subclasses. For example, the base class `EnergyModel` is a capability, but we expect many different types of energy models to be creatd and used. Each of these simply derives from the base class EnergyModel. Should the new energy model (eg. MyNewEnergy Model) need API calls not defined in the base class, the node subclass using that model can define it's own type specific `Get` method `MyNewEnergyModel* GetMyNewEnergy())` to return the model in a typesafe manner.

To create a new node class, perform the following steps.

1. Create your node subclass as a direct descendent of the Node base class.

2. Add members to your node subclass that are pointers to each of the node capabilities you need. We use pointers here rather than direct objects, since you might want a SensorNode with a specific energy model that derives from the base EnergyModel capability.
3. Override each of the "Get*" virtual member functions of the Node base class to return the appropriate pointer to each capability.
4. Implement a copy constructor that calls the "Copy" method on each capability in your class. Do NOT just copy the pointers, as this will result in "double delete".
5. Implement a destructor that deletes each of your capabilities.
6. Implement a Copy() method that returns a copy of your node. This is usually just one line of code, calling "new" and specifying the copy constructor.

To implement a new Capability, perform the following steps:

1. Create your new capability class as a direct descendent of the NodeCapability base class.
2. If needed, implement a copy constructor. This is typically only needed if your capability does dynamic memory management (ie. new and delete).
3. If needed, implement a destructor. Again, this is typically only needed if you use dynamic memory.
4. Implement a Copy() method that returns a copy of your capability.

To implement a variation on an existing capability, perform the following steps:

1. Create your new capability as a subclass of an existing capability.
2. Override the capability members as needed to implement the desired behavior.
3. Override the Copy() method to create a copy of your capability subclass.

The design team for *ns-3* expects that the number of different node capabilities will remain relatively small over time. Contributors and those modifying ns3 for their own uses are encouraged to subclass an existing capability where possible.

8.1.3 Layered architecture

explain how protocol layers are interconnected and go through standard objects to pass packets up and down the stack. Provide a diagram.

8.2 Applications/Sockets Interface

The interface between the Applications and “what lies below” is incomplete in this release. It is generally agreed that some variant of a sockets-like API will allow applications to

Presently, the design is as follows. Each Node provides a “Getter” function to an ApplicationList that stores a pointer to all applications on a node.

```
  virtual ApplicationList* GetApplicationList() const;
```

This ApplicationList will delete the applications upon deletion of the nodes, or when the application completes and calls the `Done` method in the ApplicationList.

There is a base class Application. The comments on this class read:

```
// Class Application is the base class for all ns3 applications.
// Applications are associated with individual nodes, and are created
// using the AddApplication method in the ApplicationManager capability.
//
// Conceptually, an application has zero or more Socket
// objects associated with it, that are created using the Socket
// creation API of the Kernel capability.  The Socket object
// API is modeled after the
// well-known BSD sockets interface, although it is somewhat
// simplified for use with ns3.  Further, any socket call that
// would normally "block" in normal sockets will return immediately
// in ns3.  A set of "upcalls" are defined that will be called when
// the previous blocking call would normally exit.  THis is documented
// in more detail Socket class in socket.h.
//
// There is a second application class in ns3, called "ThreadedApplication"
// that implements a true sockets interface, which should be used
// when porting existing sockets code to ns3.  The true
// sockets approach is significantly
// less memory--efficient using private stacks for each defined application,
// so that approach should be used with care.  The design and implementation
// of the ThreadedApplication are still being discussed.
```

class Application is an abstract base class. There is presently one usable application in the code: class OnOffApplication. This has the following constructor:

```
 OnOffApplication(const Node& n,
                  const Ipv4Address,  // Peer IP address
                  uint16_t,           // Peer port
                  const RandomVariable&,     // Random variable for On time
                  const RandomVariable&,     // Random variable for Off time
                  DataRate  = g_defaultRate,  // Data rate when on
                  uint32_t = g_defaultSize);  // Size of packets
```

these parameters define the behavior of an application that sends data at a certain rate and packet size for a period of time ("On") after a period of "Off" time. The destination is specified by IP address and port.

Finally, there is an instance of a DatagramSocket object implemented, which provides an object-oriented design of a UDP-like socket. We plan to support also a ServerStream and ClientStream socket that map more to TCP-like applications.

In the present code, the OnOffApplication creates a DatagramSocket and the socket later binds to a UDP endpoint. The OnOffApplication sends fake data to the sockets API, which causes packets with null payloads to be generated, or alternatively a true Send() call with application data could be sent. For socket calls that typically block in real applications, callbacks can be registered for events on the socket.

There are some open issues:

- There is no DNS-like capability to identify nodes and services by, other than IP address and port. Note that in *ns-3*, we are more fully supporting multiple interfaces per node, so it is somewhat ambiguous to just refer to a node_id like in *ns-2*.
- We would like to define a common Sockets base class if there is possibility to define applications that can take either TCP or UDP sockets and therefore could work with a common subset of the sockets API; this would make up the base class Sockets API. This is not yet implemented.
- The level of abstraction between an application and the rest of the node is to be defined. There has been a proposal to create a generic Kernel or Stack object and have that object (common across all nodes that use Applications) create the Socket and provide a pointer of socket object. There has been another proposal for an Application to explicitly provide a L4Protocol to the node, to allow fine-granularity of L4Protocol behavior (per-socket granularity).
- The Ipv4 address should be generalized (to IPv6 or others).

8.2.1 Adding Applications

The preferred method for adding a data demand to a network is through the use of *Applications*. The concept of an application in *ns-3* is conceptually identical to applications in real systems, in that applications can create and send data to other applications, receive data from other applications, respond to timers, and start or stop at any point in time. Additionally, the *ns-3* applications are very loosely modeled after the similar concept in *ns-2*.

Applications are completely managed by the `ApplicationManager` capability. They are allocated with dynamic memory, and responsiblity for managing that memory is completely the responsibility of the `ApplicationManager`. An *ns-3* application can have multiple sockets active at any time, and have simulataneous connections to many other applications.

Applications must implement a `StartApp` and `StopApp` method, to start and stop their processing. These are called by the simulation at the time specified in the `Start` and `Stop` methods, which are implemented in the application base class. Applications must also implement a `Copy` method to create a replica of itself. This is generally one line of code, as can be seen in the `OnOffApplication.cc`.

Users do not allocate dynamic memory for applications; rather they ask the `ApplicationManager` to do it as follows:

```
MyApp* myApp = node->GetApplicationManager()->Add(MyApp());
myApp->Start(Time("100ms"));
```

The `Add` method for the `ApplicationManager` creates a copy of the specified application, adds it a list of known applications for the node, and returns the pointer to the application in a typesafe manner. The argument must be a subclass of the `Application` base class. An important aspect of this design is that the user is not responsible for the memory allocated for the application. Rather the application manager will delete the memory either when the node is deleted, or the application notifies the `ApplicationManager` that it is done, using the `Done` method. Another important aspect of this design is that the applications should not do anything in the constructor that interacts with other parts of the simulator. In particular it should not allocate sockets, bind to ports, or anything that binds the appliction to other parts of *ns-3*. These should all be done in the `StartApp` method. See the code in `OnOffApplication.cc` for an example.

8.3 Stack/NetDevice Interface

This interface is the boundary between what is traditionally layer-3 (IP) and the network devices.

8.3.1 Design goals

- Device sublayer shouldn't have to know about IP addresses
- Use Linux as a template for how to design this interface

8.3.2 Design overview

The basic idea is to mimic the Linux architecture at the boundary between device-independent sublayer of the network device layer and the IP layer (figure 8.1).

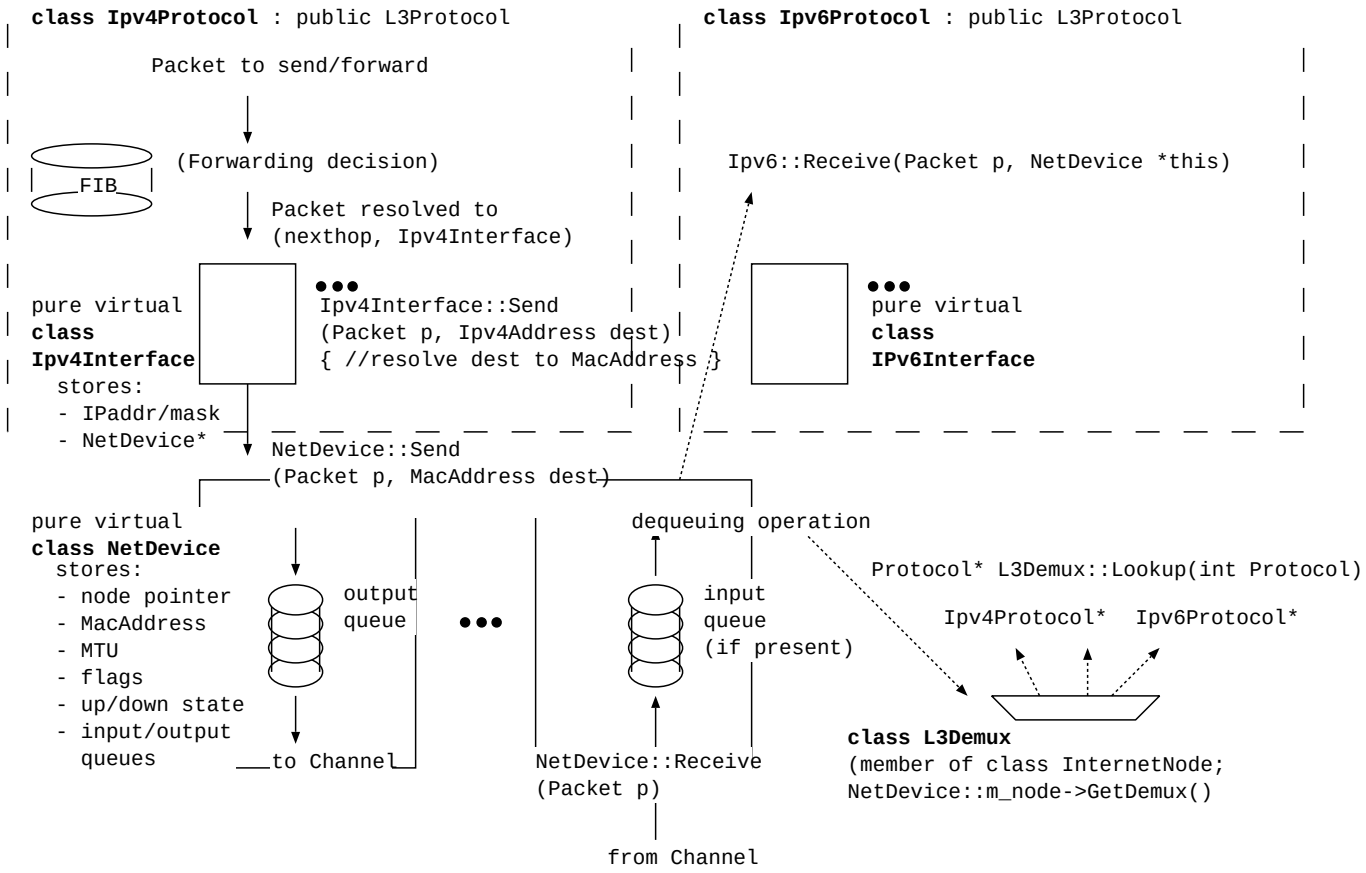

Figure 8.1: Overview of boundary between Network Device and layer-3.

Figure 8.1 illustrates some of the main objects and actions involving sending a packet up and down the stack. First, consider the sending or forwarding of a packet. We assume that the packet is an IPv4 packet being handled by an IPv4Protocol object that derives from a base class L3Protocol (which may derive from Capability class). The forwarding information base (FIB) contains the information necessary to resolve the destination address to an interface and next-hop IP address.

The interface itself is split into two components, mirroring the split of devices in Linux between a NetDevice `struct net_device` and an IPv4 portion (`struct in_device`). The class Ipv4Interface is a pure virtual class that has a function SendTo() that must be defined in a subclass. Implementations of SendTo() take the packet and resolves the next-hop address to a MAC address, append the LLC/SNAP header, etc.; it is pure virtual because depending on the link technology, different actions may be taken here (such as ARP). Ipv4Interface also contains a list of all of the IPv4 addresses on this

interface, and a pointer to an underlying NetDevice object. There is one Ipv4Interface for each NetDevice interface (plus possibly an IPv4 loopback interface), and also an Ipv6Interface for each NetDevice; that is, for each NetDevice, there is one corresponding "Protocol" device for each L3Protocol in the node.

Leaving `Ipv4Interface::SendTo()`, the packet next is received by an object of type `class NetDevice`. This boundary between layers is analogous to the point in the Linux kernel where `dev_queue_xmit()` is called; here, the packet has a valid L2 header with destination MAC address. This class is also a pure virtual base class, and depending on the link technology (802.11, 802.3, etc.) different implementations will be needed. Typically, the packet will be queued in an output queue where a queueing discipline is applied, and then the packet is passed to the lower sublayers (or other functions in the NetDevice) where actions corresponding to lower protocol layers (e.g., Linux `drivers/net/` operations) are applied. Eventually, the packet is sent on some type of Channel object. The NetDevice contains a node pointer, an ifIndex and string name (such as "eth0"), a MacAddress, a MTU, "flags" that describe the capabilities of the interface, up/down state, and input/output queues. In Linux, this queue may be subject to QoS and traffic control disciplines, and we intend to provide similar configuration/capability here (e.g., the interface for configuring the queues may align somewhat with Linux "tc" tool0.

Next, we discuss packets traversing the stack in the receive direction. At some point, they are processed by the NetDevice again, and possibly subjected to input queueing. At the end of the layer-2 processing, the packet must be delivered to the correct higher layer protocol. Here, there is an explicit demultiplexer (member of `class InternetNode`) used. Each Protocol above is registered with this Demux object, and the NetDevice looks up the right Protocol object with a lookup of the protocol number. With this Protocol object, pointer, the NetDevice calls a virtual Receive() function on the Protocol.

Some initial code is found in the following repository http://code.nsnam.org/tomh/ns-3:

- `src/node/` directory contains the source files
- `samples/main-net-device-if.cc` is the start of a sample program to demonstrate the code

This code compiles and has some doxygen comments but is still a work in progress and may not line up 100% with the above explanation.

Items still left to deal with:

- NetDevice needs an accessor to add MAC multicast addresses
- NetDevice needs a set of getters to test for the broadcast/multicast/unicast bools.
- NetDevice needs a way for the subclasses to set the broadcast/multicast/unicast bools

In closing, there is a desire to define commonality between various subclasses of both Channels and NetDevices. However, at the moment, it seems like only the "top" interface (with L3Protocol) is general enough to be common across a wide class of NetDevices. The interface between NetDevice and Channel is likely to be technology specific, and the configuration interface for NetDevice will likely have a lot of interface aspects defined in the subclasses only.

8.3.3 Configuration of Ipv4Interface

TODO.

8.3.4 Configuration of NetDevice, including Queues

TODO.

8.4 NetDevice/Channel Interface

Channels are objects that interconnect NetDevice objects. The NetDevice objects are owned, from a memory management perspective, by each node. The Channel is a reference counted object, and each NetDevice on the Channel has a reference to the Channel.

Base class Channel is pure virtual. It was only recently that there was agreement that a base class Channel is needed in *ns-3*. Today, there is still uncertainty over how much functionality will be in a base class vs. being specialized. It has been thought that the minimal functionality across all channels was a method to find all of the NetDevice objects attached to it. This is implemented by the following two Channel member functions:

```
virtual uint32_t GetNDevices (void) const = 0;
virtual NetDevice *GetDevice (uint32_t i) const = 0;
```

It is thought that, in general, there will be less modularity of components at this interface; e.g., an Ethernet NetDevice will be connected to an Ethernet channel only, although it may be that there will be modularity between NetDevices of very similar technology or designed for different levels of abstraction.

In this release, src/devices/p2p/ contains an implementation of a generic PointToPointNetDevice and PointToPointChannel.

Chapter 9

Packets

Packets (messages) are fundamental objects in the simulator and their design is important from a performance and resource management perspective. There are various ways to design the simulation packet, and tradeoffs among the different approaches. In particular, there is a tension between ease-of-use, performance, and safe interface design.

There are a few requirements on this object design:

- Creation, management, and deletion of this object should be as simple as possible, while avoiding the chance for memory leaks and/or heap corruption;
- Packets should support serialization and deserialization so that network emulation is supported;
- Packets should support fragmentation and concatenation (multiple packets in a data link frame), especially for wireless support;
- It should be natural for packets to carry actual application data, or if there is only an emulated application and there is no need to carry dummy bytes, smaller packets could be used with just the headers and a record of the payload size, but not actual application bytes, conveyed in the simulated packet.
- Packets should facilitate BSD-like operations on mbufs, for support of ported operating system stacks.
- Additional side-information should be supported, such as a tag for cross-layer information.

9.1 Packet design overview

Unlike *ns-3*2, in which Packet objects contain a buffer of C++ structures corresponding to protocol headers, each network packet in *ns-3* contains a byte Buffer and a list of Tags:

- The byte buffer stores the serialized content of the chunks added to a packet. The serialized representation of these chunks is expected to match that of real network packets bit for bit (although nothing forces you to do this) which means that the content of a packet buffer is expected to be that of a real packet. Packets can also be created with an arbitrary zero-filled payload for which no real memory is allocated.
- The list of tags stores an arbitrarily large set of arbitrary user-provided data structures in the packet. Each Tag is uniquely identified by its type; only one instance of each type of data structure is allowed in a list of tags. These tags typically contain per-packet cross-layer information or flow identifiers (i.e., things that you wouldn't find in the bits on

the wire). Each tag stored in the tag list can be at most 16 bytes. Trying to attach bigger data structures will trigger crashes at runtime. The 16 byte limit is a modifiable compilation constant.

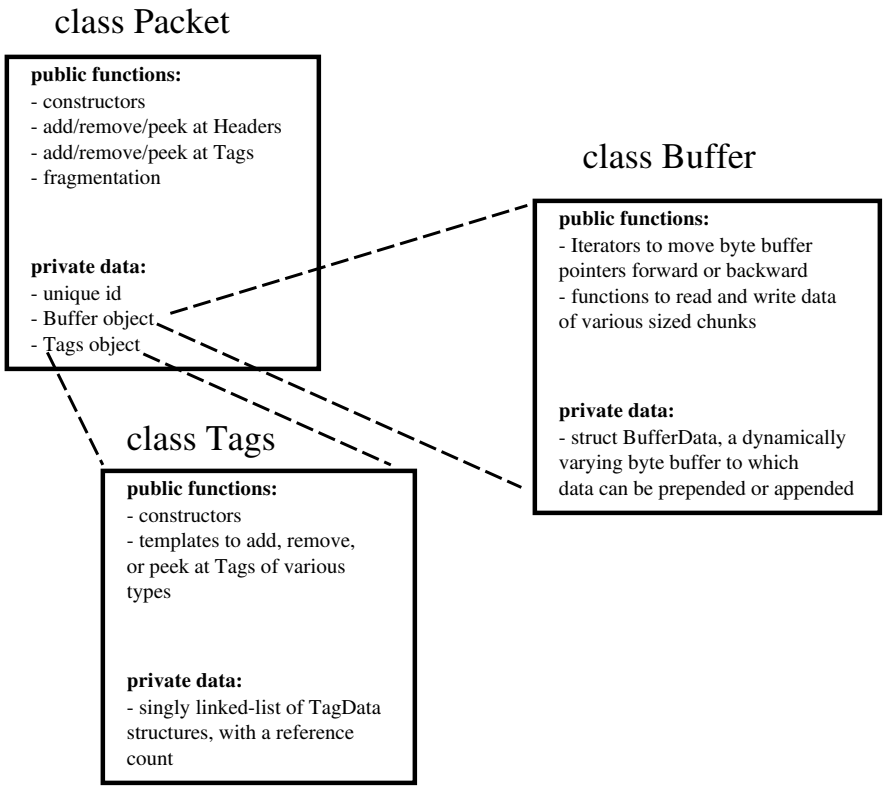

Figure 9.1: Implementation overview of Packet class.

Figure 9.1 is a high-level overview of the Packet implementation; more detail on the byte Buffer implementation is provided later in Figure 9.2. In *ns-3*, the Packet byte buffer is analogous to a Linux skbuff or BSD mbuf; it is a serialized representation of the actual data in the packet. The tag list is a container for extra items useful for simulation convenience; if a Packet is converted to an emulated packet and put over an actual network, the tags are stripped off and the byte buffer is copied directly into a real packet.

The Packet class has value semantics: it can be freely copied around, allocated on the stack, and passed to functions as arguments. Whenever an instance is copied, the full underlying data is not copied; it has "copy-on-write" (COW) semantics. Packet instances can be passed by value to function arguments without any performance hit.

The fundamental classes for adding to and removing from the byte buffer are `class Header` and `class Trailer`. Headers are more common but the below discussion also largely applies to protocols using trailers. Every protocol header that needs to be inserted and removed from a Packet instance should derive from the abstract Header base class and implement the private pure virtual methods listed below:

- `ns3::Header::SerializeTo()`
- `ns3::Header::DeserializeFrom()`
- `ns3::Header::GetSerializedSize()`
- `ns3::Header::PrintTo()`

Basically, the first three functions are used to serialize and deserialize protocol control information to/from a Buffer. For example, one may define `class TCPHeader : public Header`. The TCPHeader object will typically consist of some private data (like a sequence number) and public interface access functions (such as checking the bounds of an input). But the underlying representation of the TCPHeader in a Packet Buffer is 20 serialized bytes (plus TCP options). The TCPHeader::SerializeTo() function would therefore be designed to write these 20 bytes properly into the packet, in network byte order. The last function is used to define how the Header object prints itself onto an output stream.

Similarly, user-defined Tags can be appended to the packet. Unlike Headers, Tags are not serialized into a contiguous buffer but are stored in an array. By default, Tags are limited to 16 bytes in size. Tags can be flexibly defined to be any type, but there can only be one instance of any particular object type in the Tags buffer at any time. The implementation makes use of templates to generate the proper set of Add(), Remove(), and Peek() functions for each Tag type.

9.2 Packet interface

The public member functions of a Packet object are as follows:

9.2.1 Constructors

```
/**
 * Create an empty packet with a new uid (as returned
 * by getUid).
 */
Packet ();
/**
 * Create a packet with a zero-filled payload.
 * The memory necessary for the payload is not allocated:
 * it will be allocated at any later point if you attempt
 * to fragment this packet or to access the zero-filled
 * bytes. The packet is allocated with a new uid (as
 * returned by getUid).
 *
 * \param size the size of the zero-filled payload
 */
Packet (uint32_t size);
```

9.2.2 Adding and removing Buffer data

The below code is reproduced for Header class only; similar functions exist for Trailers.

```
/**
 * Add header to this packet. This method invokes the
 * ns3::Header::serializeTo method to request the header to serialize
 * itself in the packet buffer.
 *
 * \param header a reference to the header to add to this packet.
 */
void Add (Header const &header);
```

```
/**
 * Deserialize header from this packet. This method invokes the
 * ns3::Header::deserializeFrom method to request the header to deserialize
 * itself from the packet buffer. This method does not remove
 * the data from the buffer. It merely reads it.
 *
 * \param header a reference to the header to deserialize from the buffer
 */
void Peek (Header &header);
/**
 * Remove a deserialized header from the internal buffer.
 * This method removes the bytes read by Packet::peek from
 * the packet buffer.
 *
 * \param header a reference to the header to remove from the internal buffer.
 */
void Remove (Header const &header);
/**
 * Add trailer to this packet. This method invokes the
 * ns3::Trailer::serializeTo method to request the trailer to serialize
 * itself in the packet buffer.
 *
 * \param trailer a reference to the trailer to add to this packet.
 */
```

9.2.3 Adding and removing Tags

```
/**
 * Attach a tag to this packet. The tag is fully copied
 * in a packet-specific internal buffer. This operation
 * is expected to be really fast.
 *
 * \param tag a pointer to the tag to attach to this packet.
 */
template <typename T>
void AddTag (T const &tag);
/**
 * Remove a tag from this packet. The data stored internally
 * for this tag is copied in the input tag if an instance
 * of this tag type is present in the internal buffer. If this
 * tag type is not present, the input tag is not modified.
 *
 * This operation can be potentially slow and might trigger
 * unexpectedly large memory allocations. It is thus
 * usually a better idea to create a copy of this packet,
 * and invoke removeAllTags on the copy to remove all
 * tags rather than remove the tags one by one from a packet.
 *
 * \param tag a pointer to the tag to remove from this packet
 * \returns true if an instance of this tag type is stored
 *          in this packet, false otherwise.
 */
```

```
template <typename T>
bool RemoveTag (T &tag);
/**
 * Copy a tag stored internally to the input tag. If no instance
 * of this tag is present internally, the input tag is not modified.
 *
 * \param tag a pointer to the tag to read from this packet
 * \returns true if an instance of this tag type is stored
 *          in this packet, false otherwise.
 */
template <typename T>
bool PeekTag (T &tag) const;
/**
 * Remove all the tags stored in this packet. This operation is
 * much much faster than invoking removeTag n times.
 */
void RemoveAllTags (void);
```

9.2.4 Fragmentation

```
/**
 * Create a new packet which contains a fragment of the original
 * packet. The returned packet shares the same uid as this packet.
 *
 * \param start offset from start of packet to start of fragment to create
 * \param length length of fragment to create
 * \returns a fragment of the original packet
 */
Packet CreateFragment (uint32_t start, uint32_t length) const;

    /**
     * Concatenate the input packet at the end of the current
     * packet. This does not alter the uid of either packet.
     *
     * \param packet packet to concatenate
     */
void addAtEnd (Packet packet);

    /oncatenate the input packet at the end of the current
 * packet. This does not alter the uid of either packet.
 *
 * \param packet packet to concatenate
 */
void AddAtEnd (Packet packet);
/**
 * Concatenate the fragment of the input packet identified
 * by the offset and size parameters at the end of the current
 * packet. This does not alter the uid of either packet.
 *
 * \param packet to concatenate
 * \param offset offset of fragment to copy from the start of the input packet
 * \param size size of fragment of input packet to copy.
```

```
 */
void AddAtEnd (Packet packet, uint32_t offset, uint32_t size);
/**
 * Remove size bytes from the end of the current packet
 * It is safe to remove more bytes that what is present in
 * the packet.
 *
 * \param size number of bytes from remove
 */
void RemoveAtEnd (uint32_t size);
/**
 * Remove size bytes from the start of the current packet.
 * It is safe to remove more bytes that what is present in
 * the packet.
 *
 * \param size number of bytes from remove
 */
void RemoveAtStart (uint32_t size);
```

9.2.5 Miscellaneous

```
/**
 * \returns the size in bytes of the packet (including the zero-filled
 *          initial payload)
 */
uint32_t GetSize (void) const;
/**
 * If you try to change the content of the buffer
 * returned by this method, you will die.
 *
 * \returns a pointer to the internal buffer of the packet.
 */
uint8_t const *PeekData (void) const;
/**
 * A packet is allocated a new uid when it is created
 * empty or with zero-filled payload.
 *
 * \returns an integer identifier which uniquely
 *          identifies this packet.
 */
uint32_t GetUid (void) const;
```

9.3 Using Headers

walk through an example of adding a UDP header

9.4 Using Tags

walk through an example of adding a flow ID

9.5 Using Fragmentation

walk through an example of link-layer fragmentation/reassembly

9.6 Sample program

The below sample program (from `ns3/samples/main-packet.cc` illustrates some use of the Packet, Header, and Tag classes.

```
/* -*-    Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:nil -*- */
#include "ns3/packet.h"
#include "ns3/header.h"
#include <iostream>

using namespace ns3;

/* A sample Header implementation
 */
class MyHeader : public Header {
public:
    MyHeader ();
    virtual ~MyHeader ();

    void SetData (uint16_t data);
    uint16_t GetData (void) const;
private:
    virtual void PrintTo (std::ostream &os) const;
    virtual void SerializeTo (Buffer::Iterator start) const;
    virtual void DeserializeFrom (Buffer::Iterator start);
    virtual uint32_t GetSerializedSize (void) const;

    uint16_t m_data;
};

MyHeader::MyHeader ()
{}
MyHeader::~MyHeader ()
{}
void
MyHeader::PrintTo (std::ostream &os) const
{
    os << "MyHeader data=" << m_data << std::endl;
}
```

```
uint32_t
MyHeader::GetSerializedSize (void) const
{
    return 2;
}
void
MyHeader::SerializeTo (Buffer::Iterator start) const
{
    // serialize in head of buffer
    start.WriteHtonU16 (m_data);
}
void
MyHeader::DeserializeFrom (Buffer::Iterator start)
{
    // deserialize from head of buffer
    m_data = start.ReadNtohU16 ();
}

void
MyHeader::SetData (uint16_t data)
{
    m_data = data;
}
uint16_t
MyHeader::GetData (void) const
{
    return m_data;
}

/* A sample Tag implementation
 */
struct MyTag {
    uint16_t m_streamId;
};

static TagRegistration<struct MyTag> g_MyTagRegistration ("ns3::MyTag", 0);


static void
Receive (Packet p)
{
    MyHeader my;
    p.Peek (my);
    p.Remove (my);
    std::cout << "received data=" << my.GetData () << std::endl;
    struct MyTag myTag;
    p.PeekTag (myTag);
}


int main (int argc, char *argv[])
{
    Packet p;
    MyHeader my;
```

```
    my.SetData (2);
    std::cout << "send data=2" << std::endl;
    p.Add (my);
    struct MyTag myTag;
    myTag.m_streamId = 5;
    p.AddTag (myTag);
    Receive (p);
    return 0;
}
```

9.7 Implementation details

9.7.1 Private member variables

A Packet object's interface provides access to some private data:

```
    Buffer m_buffer;
    Tags m_tags;
    uint32_t m_uid;
    static uint32_t m_global_uid;
```

Each Packet has a Buffer and a Tags object, and a 32-bit unique ID (m_uid). A static member variable keeps track of the UIDs allocated. Note that real network packets do not have a UID; the UID is therefore an instance of data that normally would be stored as a Tag in the packet. However, it was felt that a UID is a special case that is so often used in simulations that it would be more convenient to store it in a member variable.

9.7.2 Buffer implementation

Class Buffer represents a buffer of bytes. Its size is automatically adjusted to hold any data prepended or appended by the user. Its implementation is optimized to ensure that the number of buffer resizes is minimized, by creating new Buffers of the maximum size ever used. The correct maximum size is learned at runtime during use by recording the maximum size of each packet.

Authors of new Header or Trailer classes need to know the public API of the Buffer class. (add summary here)

The byte buffer is implemented as follows:

```
    struct BufferData {
        uint32_t m_count;
        uint32_t m_size;
        uint32_t m_initialStart;
        uint32_t m_dirtyStart;
        uint32_t m_dirtySize;
        uint8_t m_data[1];
    };
    struct BufferData *m_data;
    uint32_t m_zeroAreaSize;
```

```
uint32_t m_start;
uint32_t m_size;
```

- `BufferData::m_count`: reference count for BufferData structure
- `BufferData::m_size`: size of data buffer stored in BufferData structure
- `BufferData::m_initialStart`: offset from start of data buffer where data was first inserted
- `BufferData::m_dirtyStart`: offset from start of buffer where every Buffer which holds a reference to this BufferData instance have written data so far
- `BufferData::m_dirtySize`: size of area where data has been written so far
- `BufferData::m_data`: pointer to data buffer
- `Buffer::m_zeroAreaSize`: size of zero area which extends before `m_initialStart`
- `Buffer::m_start`: offset from start of buffer to area used by this buffer
- `Buffer::m_size`: size of area used by this Buffer in its BufferData structure

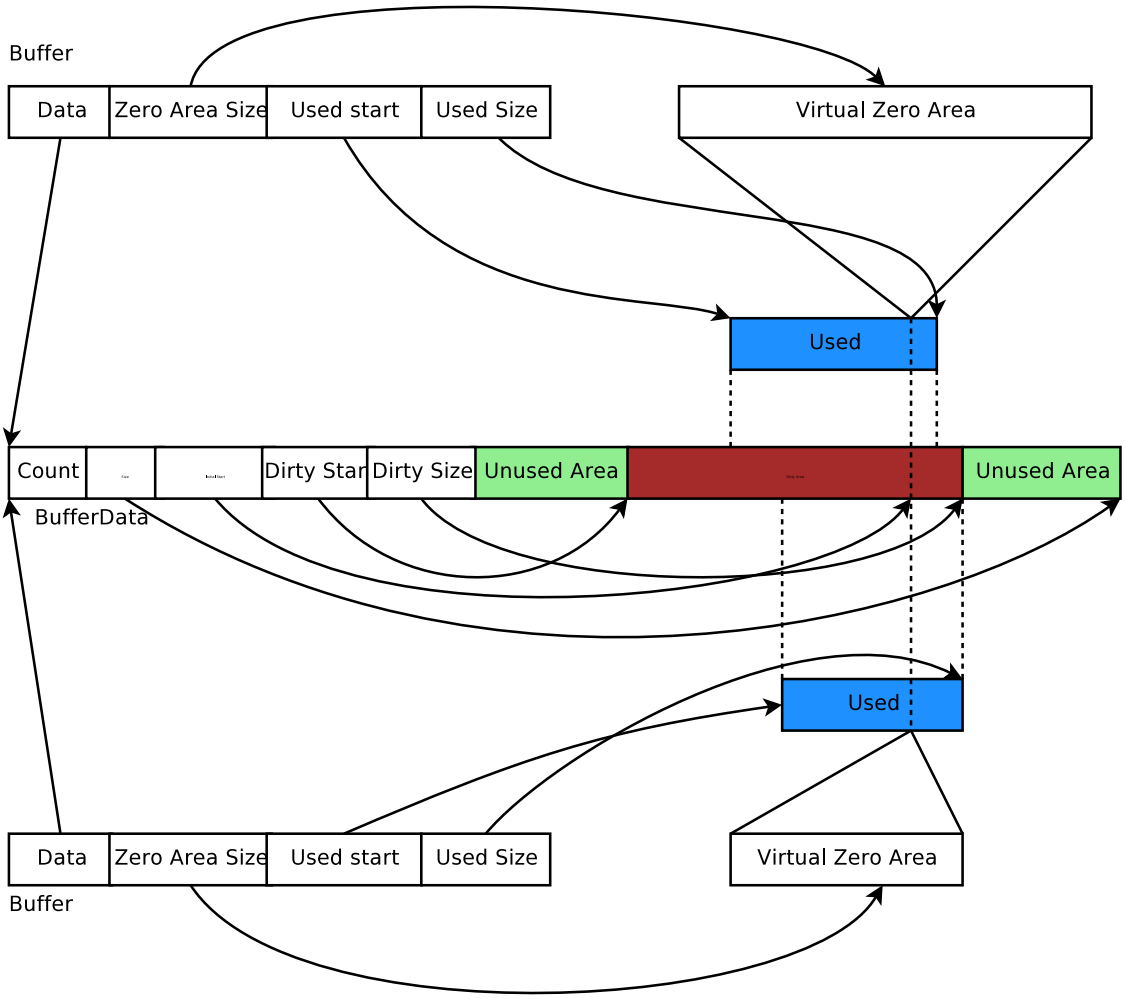

Figure 9.2: Implementation overview of a packet's byte Buffer.

This data structure is summarized in Figure 9.2. Each Buffer holds a pointer to an instance of a BufferData. Most Buffers should be able to share the same underlying BufferData and thus simply increase the BufferData's reference count. If they have to change the content of a BufferData inside the Dirty Area, and if the reference count is not one, they first create a copy of the BufferData and then complete their state-changing operation.

9.7.3 Tags implementation

Tags are implemented by a single pointer which points to the start of a linked list ofTagData data structures. Each TagData structure points to the next TagData in the list (its next pointer contains zero to indicate the end of the linked list). Each TagData contains an integer unique id which identifies the type of the tag stored in the TagData.

```
struct TagData {
    struct TagData *m_next;
    uint32_t m_id;
    uint32_t m_count;
    uint8_t m_data[Tags::SIZE];
};
class Tags {
    struct TagData *m_next;
};
```

Adding a tag is a matter of inserting a new TagData at the head of the linked list. Looking at a tag requires you to find the relevant TagData in the linked list and copy its data into the user data structure. Removing a tag and updating the content of a tag requires a deep copy of the linked list before performing this operation. On the other hand, copying a Packet and its tags is a matter of copying the TagData head pointer and incrementing its reference count.

Tags are found by the unique mapping betweent the Tag type and its underlying id. This is why at most one instance of any Tag can be stored in a packet. The mapping between Tag type and underlying id is performed by a registration as follows:

```
/* A sample Tag implementation
 */
struct MyTag {
    uint16_t m_streamId;
};
```

add description of TagRegistration for printing

9.7.4 Memory management

Describe free list.

Describe dataless vs. data-full packets.

9.7.5 Copy-on-write semantics

The current implementation of the byte buffers and tag list is based on COW (Copy On Write). An introduction to COW can be found in Scott Meyer's "More Effective C++", items 17 and 29). This design feature and aspects of the public interface

borrows from the packet design of the Georgia Tech Network Simulator. This implementation of COW uses a customized reference counting smart pointer class.

What COW means is that copying packets without modifying them is very cheap (in terms of CPU and memory usage) and modifying them can be also very cheap. What is key for proper COW implementations is being able to detect when a given modification of the state of a packet triggers a full copy of the data prior to the modification: COW systems need to detect when an operation is "dirty" and must therefore invoke a true copy.

Dirty operations:

- Packet::RemoveTag()
- Packet::Add()
- both versions of ns3::Packet::AddAtEnd()

Non-dirty operations:

- Packet::AddTag()
- Packet::RemoveAllTags()
- Packet::PeekTag()
- Packet::Peek()
- Packet::Remove()
- Packet::CreateFragment()
- Packet::RemoveAtStart()
- Packet::RemoveAtEnd()

Dirty operations will always be slower than non-dirty operations, sometimes by several orders of magnitude. However, even the dirty operations have been optimized for common use-cases which means that most of the time, these operations will not trigger data copies and will thus be still very fast.

Chapter 10

Channel

Disclaimer: This section and underlying implementation is very incomplete.

A channel is used to interconnect nodes in *ns-3*. Specifically, a channel interconnects NetDevices on *ns-3* nodes.

It is not yet clear which functions should properly reside in a Channel base class, vs. within a subclass. Presently, the only functions in the base class are those used to count the number of NetDevices attached to the channel, and an accessor function to get a NetDevice pointer to the indexed NetDevice.

A printout of the base class Channel interface (ns-3/src/node/channel.h) is shown below; a subclass of this channel (PointTo-PointChannel) can be found in the source code in the directory ns-3/src/devices/p2p/.

```
#ifndef CHANNEL_H
#define CHANNEL_H

#include <string>
#include <stdint.h>

namespace ns3 {

class NetDevice;

/**
 * \brief Abstract Channel Base Class.
 *
 * A channel is a logical path over which information flows.  The path can
 * be as simple as a short piece of wire, or as complicated as space-time.
 */
class Channel
{
public:
  Channel ();
  Channel (std::string name);
  virtual ~Channel ();

  void SetName(std::string);
```

```
  std::string GetName(void);

  virtual uint32_t GetNDevices (void) const = 0;
  virtual NetDevice *GetDevice (uint32_t i) const = 0;

protected:
  std::string m_name;

private:
};

}; // namespace ns3

#endif /* CHANNEL_H */
```

Part IV

Topologies and higher level constructs

Chapter 11

Topologies

Disclaimer: This section and underlying implementation is very incomplete.

One of the design goals for *ns-3* is to allow for reusable components and topologies. For instance, there may be some number of stock topology objects, such as a wireless grid, a dumbbell, or a point-to-point link. Users may want to interchange the node or channel types without recreating the topologies.

For example, consider the sample topology object shown in Figure 11.0-1. This can be defined as a static function and reused by different user programs or scripts. The function allows users to specify the nodes, IP addresses, data rate, and propagation delay of a point to point link, and the function performs all of the low level functions to connect the two nodes. Similarly, *ns-3* plans to add various wired and wireless topology objects in future releases.

A challenge in using topology objects of this sort is that they tend to operate on base class pointers. Therefore, they require *virtual constructors* if the topology object needs to create objects of the right subclass of the base class pointer, and they require that the subclass be configurable via a base class interface. This presents a challenge if a user tries to use the topology object with incompatible components; a hard or soft error may result (we define a soft error here as an error in which the program does not fail at either compile or run time, but the simulation output is incorrect).

In the current codebase, the solution to these problems (described in Section 8) is the use of a virtual Copy() method as the virtual constructor for Nodes, and the definition of virtual Capabilities() as interfaces for the Node. The design team is exploring additional solutions to this problem.

```
1  P2PChannel* Topology::AddDuplexLink(Node* n1, const IPAddr& ip1,
2                                      Node* n2, const IPAddr& ip2,
3                                      const Rate& rate, const Time& delay)
4  {
5    // First get the NetDeviceList capability from each node
6    NetDeviceList* ndl1 = n1->GetNetDeviceList();
7    NetDeviceList* ndl2 = n2->GetNetDeviceList();
8    if (!ndl1 || !ndl2) return nil;  // Both ends must have NetDeviceList
9    // Get the net devices
10   P2PNetDevice* nd1 = ndl1->Add(P2PNetDevice(n1, rate, nil));
11   P2PNetDevice* nd2 = ndl2->Add(P2PNetDevice(n1, rate, nd1->GetChannel()));
12   // Not implemented yet.  Add appropriate layer 2 protocol for
13   // the net devices.
14   // Get the L3 proto for node 1 and configure it with this device
15   L3Demux*    l3demux1 = n1->GetL3Demux();
16   L3Protocol* l3proto1 = nil;
17   // If the node 1 l3 demux exists, find the coresponding l3 protocol
18   if (l3demux1) l3proto1 = l3demux1->Lookup(ip1.L3Proto());
19   // If the l3 protocol exists, configure this net device.  Use a mask
20   // of all ones, since there is only one device on the remote end
21   // of this link
22   if (l3proto1) l3proto1->AddNetDevice(nd1, ip1, ip1.GetMask(ip1.Size()*8));
23   // Same for node 2
24   L3Demux*    l3demux2 = n2->GetL3Demux();
25   L3Protocol* l3proto2 = nil;
26   // If the node 2 l3 demux exists, find the coresponding l3 protocol
27   if (l3demux2) l3proto2 = l3demux2->Lookup(ip2.L3Proto());
28   if (l3proto2) l3proto2->AddNetDevice(nd2, ip2, ip2.GetMask(ip2.Size()*8));
29   return dynamic_cast<P2PChannel*>(nd1->GetChannel());  // Always succeeds
30 }
```

Program 11.0-1 topology.cc

Part V

Support

Chapter 12

Tracing and Logging Implementation Overview

12.1 Design Overview: Tracing and Callbacks

The Tracing Framework is built around two levels of API:

- a low-level API which is used to generate arbitrary trace events and connect them to arbitrary trace handlers
- a high-level API which provides a set of default trace handlers which generate trace files and hide all of the details of trace setup behind a conveniant API.

12.1.1 The high-level API

The high-level API is not very well defined right now: we expect it to be updated to handle as many realistic use-cases as possible. For now, the high-level API is built around the AsciiTrace class which generates a text-based trace file from all MAC-level queue and receive events in all nodes.

12.1.2 The low-level API

The purpose of the low-level API is to be as flexible as possible to allow the user to build advanced tracing analysis mechanisms or to allow us to build any kind of high-level API. The low-level API thus enforces no policy: it merely provides a mechanism to generate various kinds of events and connect these event sources to user-provided event handlers.

This API is built around a few concepts:

- There can be any number of trace source objects. Each trace source object can generate any number of trace events. The current trace source objects are: ns3::CallbackTraceSourceSource, ns3::UVTraceSource, ns3::SVTraceSource, and, ns3::FVTraceSource.
- Each trace source can be connected to any number of trace sinks. A trace sink is a ns3::Callback (see section 6.4) with a very special signature. Its first argument is always a ns3::TraceContext.

- Every trace source is uniquely identified by a ns3::TraceContext. Every trace sink can query a ns3::TraceContext for information. This allows a trace sink which is connected to multiple trace sources to identify from which source each event is coming from.

To define new trace sources, a model author needs to instanciate one trace source object for each kind of tracing event he wants to export. The trace source objects currently defined are:

- ns3::CallbackTraceSourceSource: this trace source can be used to convey any kind of trace event to the user. It is a functor, that is, it is a variable which behaves like a function which will forward every event to every connected trace sink (i.e., ns3::Callback). This trace source takes up to four arguments and forwards these 4 arguments together with the ns3::TraceContext which identifies this trace source to the connected trace sinks.
- ns3::UVTraceSource: this trace source is used to convey key state variable changes to the user. It behaves like a normal integer unsigned variable: you can apply every normal arithmetic operator to it. It will forward every change in the value of the variable back to every connected trace sink by providing a TraceContext, the old value and the new value.
- ns3::SVTraceSource: this is the signed integer equivalent of ns3::UVTraceSource.
- ns3::FVTraceSource: this is the floating point equivalent of ns3::UVTraceSource and ns3::SVTraceSource.

For example, to define a trace source which notifies you of a new packet being transmitted, you would have to:

```
class MyModel
{
  void Tx (Packet const &p);
private:
  CallbackTraceSource<Packet const &> m_txTrace;
};

void
MyModel::Tx (Packet const &p)
{
  // trace packet tx event.
  m_txTrace (p);
  // ... send the packet for real.
}
```

Once the model author has instantiated these objects and has wired them in his simulation code (that is, he calls them wherever he wants to trigger a trace event), he needs to make these trace sources available to users to allow them to connect any number of trace sources to any number of user trace sinks. While it would be possible to make each model export directly each of his trace source instances and request users to invoke a source->Connect (callback) method to perform the connection explicitely, it was felt that this was a bit cumbersome to do.

As such, the “connection” between a set of sources and a sink is performed through a third-party class, the TraceResolver, which can be used to automate the connection of multiple matching trace sources to a single sink. This TraceResolver works by defining a hierarchical tracing namespace: the root of this namespace is accessed through the ns3::TraceRoot class. The namespace is represented as a string made of multiple elements, each of which is separated from the other elements by the ’/’ character. A namespace string always starts with a ’/’.

By default, the current simulation models provide a ’/nodes’ tracing root. This ’/nodes’ namespace is structured as follows:

```
/nodes/n/udp
```

```
/nodes/n/arp
/nodes/n/ipv4
            /tx
            /rx
            /drop
            /interfaces/n/netdevice
                                /queue/
                                      /enque
                                      /deque
                                      /drop
```

The 'n' element which follows the /nodes and /interfaces namespace elements identify a specific node and interface through their index within the ns3::NodeList and ns3::Ipv4 objects respectively.

To connect a trace sink to a trace source identified by a namespace string, a user can call the ns3::TraceRoot::Connect method (the ns3::TraceRoot::Disconnect method does the symmetric operation). This connection method can accept fully-detailed namespace strings but it can also perform pattern matching on the user-provided namespace strings to connect multiple trace sources to a single trace sink in a single connection operation.

The syntax of the pattern matching rules are loosely based on regular expressions:

- the '*' character matches every element
- the (a|b) construct matches element 'a' or 'b'
- the [ss-ee] construct matches all numerical values which belong to the interval which includes ss and ee

For example, the user could use the following to connect a single sink to the ipv4 tx, rx, and drop trace events:

```
void MyTraceSink (TraceContext const &context, Packet &packet);
TraceRoot::Connect (''/nodes/ * /ipv4/ *'', MakeCallback (&MyTraceSink));
```

Of course, this code would work only if the signature of the trace sink is exactly equal to the signature of all the trace sources which match the namespace string (if one of the matching trace source does not match exactly, a fatal error will be triggered at runtime during the connection process). The ns3::TraceContext extra argument contains information on where the trace source is located in the namespace tree. In that example, if there are multiple nodes in this scenario, each call to the MyTraceSink function would receive a different TraceContext, each of which would contain a different NodeList::Index object.

It is important to understand exactly what an ns3::TraceContext is. It is a container for a number of type instances. Each instance of a ns3::TraceContext contains one and only one instance of a given type. ns3::TraceContext::Add can be called to add a type instance into a TraceContext instance and ns3::TraceContext::Get can be called to get a copy of a type instance stored into the ns3::TraceContext. If Get cannot retrieve the requested type, a fatal error is triggered at runtime. The values stored into an ns3::TraceContext attached to a trace source are automatically determined during the namespace resolution process. To retrieve a value from a ns3::TraceContext, the code can be as simple as this:

```
void MyTraceSink (TraceContext const &context, Packet &packet)
{
  NodeList::Index index;
  context.Get (index);
  std::cout << ''node id='' << NodeList::GetNode (index)->GetId () << std::endl;
}
```

The hierarchical global namespace described here is not implemented in a single central location: it was felt that doing this would make it too hard to introduce user-specific models which could hook automatically into the overal tracing system. If the tracing namespace was implemented in a single central location, every model author would have had to modify this central component to make his own model available to trace users.

Instead, the handling of the namespace is distributed across every relevant model: every model implements only the part of the namespace it is really responsible for. To do this, every model is expected to provide an instance of a TraceResolver whose responsability is to recursively provide access to the trace sources defined in its model. Each TraceResolver instance should be a subclass of the TraceResolver base class which implements either the DoLookup or the DoConnect and DoDisconnect methods. Because implementing these methods can be a bit tedious, our tracing framework provides a number of helper template classes which should save the model author from having to implement his own in most cases:

- ns3::CompositeTraceResolver: this subclass of ns3::TraceResolver can be used to aggregate together multiple trace sources and multiple other ns3::TraceResolver instances.
- ns3::ArrayTraceResolver: this subclass of ns3::TraceResolver can be used to match any number of elements within an array where every element is identified by its index.

Once you can instantiate your own ns3::TraceResolver object instance, you have to hook it up into the global namespace. There are two ways to do this:

- you can hook your ns3::TraceResolver creation method as a new trace root by using the ns3::TraceRoot::Register method
- you can hook your new ns3::TraceResolver creation method into the container of your model. This step will obvsiouly depend on which model contains your own model but, if you wrote a new l3 protocol, all you would have to do to hook into your container L3Demux class is to implement the pure virtual method inherited from the L3Protocol class whose name is ns3::L3protocol::CreateTraceResolver.

So, in most cases, exporting a model's trace sources is a matter of implementing a method CreateTraceResolver as shown below:

```
class MyModel
{
public:
  enum TraceType {
    TX,
    RX,
    ...
  };
  TraceResolver *CreateTraceResolver (TraceContext const &context);
  void Tx (Packet const &p);
private:
  CallbackTraceSource<Packet const &> m_txTrace;
};

TraceResolver *
MyModel::CreateTraceResolver (TraceContext const &context)
{
  CompositeTraceResolver *resolver = new CompositeTraceResolver (context);
  resolver->Add (''tx'', m_txTrace, MyModel::TX);
  return resolver;
```

```
}
void
MyModel::Tx (Packet const &p)
{
  m_txTrace (p);
}
```

If you really want to have fun and implement your own ns3::TraceResolver subclass, you need to understand the basic Connection and Disconnection algorithm. The code of that algorithm is wholy contained in the ns3::TraceResolver::Connect and ns3::TraceResolver::Disconnect methods. The idea is that we recursively parse the input namespace string by removing the first namespace element. This element is 'resolved' is calling the ns3::TraceResolver::DoLookup method which returns a list of TraceResolver instances. Each of the returned TraceResolver instance is then given what is left of the namespace by calling ns3::TraceResolver::Connect until the last namespace element is processed. At this point, we invoke the ns3::TraceResolver::DoConnect or ns3::TraceResolver::DoDisconnect methods to break the recursion. A good way to understand this algorithm is to trace its behavior. Let's say that you want to connect to '/nodes/*/ipv4/interfaces/*/netdevice/queue/*'. It would generate the following call traces:

```
TraceRoot::Connect (''/nodes/*/ipv4/interfaces/*/netdevice/queue/*'', callback);
traceContext = TraceContext ();
rootResolver = CompositeTraceResolver (traceContext);
rootResolver->Connect (''/nodes/*/ipv4/interfaces/*/netdevice/queue/*'', callback);
  resolver = CompositeTraceResolver::DoLookup (''nodes'');
    return NodeList::CreateTraceResolver (GetContext ());
      return ArrayTraceResolver (context);
  resolver->Connect (''/*/ipv4/interfaces/*/netdevice/queue/*'', callback);
    ArrayTraceResolver::DoLookup (''*'');
      for (i = 0; i < n_nodes; i++)
        resolver = nodes[i]->CreateTraceResolver (GetContext ());
          return CompositeTraceResolver (context);
        resolvers.add (resolver);
      return resolvers;
    for resolver in (resolvers)
      resolver->Connect (''/ipv4/interfaces/*/netdevice/queue/*'', callback);
        CompositeTraceResolver::DoLookup (''ipv4'');
          resolver = ipv4->CreateTraceResolver (GetContext ());
            return CompositeTraceResolver (context);
          return resolver;
        resolver->Connect (''/interfaces/*/netdevice/queue/*'', callback);
          CompositeTraceResolver::DoLookup (''interfaces'');
            resolver = ArrayTraceResolver (GetContext ());
          resolver->Connect (''/*/netdevice/queue/*'', callback);
            ArrayTraceResolver::DoLookup (''*'');
              for (i = 0; i < n_interfaces; i++)
                 resolver = interfaces[i]->CreateTraceResolver (GetContext ());
                   return CompositeTraceResolver ()
                 resolvers.add (resolver);
            return resolvers;
          resolver->Connect (''/netdevice/queue/*'', callback);
            CompositeTraceResolver::DoLookup (''netdevice'');
              resolver = NetDevice::CreateTraceResolver (GetContext ());
                return CompositeTraceResolver ();
              return resolver;
            resolver->Connect (''/queue/*'', callback);
```

```
CompositeTraceResolver::DoLookup (‘‘queue’’);
  resolver = Queue::CreateTraceResolver (GetContext ());
    return CompositeTraceResolver ();
  return resolver
resolver->Connect (‘‘*’’, callback);
  CompositeTraceResolver::DoLookup (‘‘*’’);
    for match in (matches)
      resolver = TerminalTraceResolver (‘‘match’’);
      resolvers.add (resolver)
    return resolvers;
  for resolver in (resolvers)
    TerminalTraceResolver->DoConnect (callback);
```

Chapter 13

Statistics

Chapter 14

Random variables

14.1 Design Overview and Motivation

Meaningful simulations often require the modelling of stochastic processes, such as traffic patterns on a network or channel noise and interference. As such, *ns-3* provides a rich set of random number generators (RNGs) to help users model such stochastic processes. These are configured as pseudo-random number generators implementing the combined multiple recursive generator MRG32k3a proposed by L'Ecuyer in 1999. Our RNGs draw from the Uniform(0,1) distribution provided by the MRG32k3a, and then perform an inverse cumulative distribution transform to effectively sample from the target distribution in all cases except for the normal distribution, in which case the Box-Muller transform is used.

The class structure is a set of classes that derive from a sinlge class called `RandomVariable`. `RandomVariable` defines the API for all RNGs, with methods such as `GetValue()` which returns a random number from the underlying statistical distribution. `RandomVariable` also has some static methods to allow for configuration of global seeding behavior of RNGs.

14.2 Supported Distributions

At present, *ns-3* supports the following statistical distributions:

- Exponential distribution
- Normal distribution
- Pareto distribution
- Uniform distribution
- Weibull distribution
- Arbitrary distributions with user specified CDF

In addition, there are other generators that can be treated in the simulator like RNGs, but which in fact give very predictable values:

- Constant variable- Returns the same value each time a value is requested.
- Deterministic variable - Returns values from a user-defined list.
- Sequential variable - Returns values from a user-defined monotonically increasing sequence.

14.2.1 Example

```
ExponentialVariable timeDelay(1.2);
UniformVariable u01(0,1);
//etc...
```

The API docs generated by Doxygen have more specific information.

14.3 Seeding

There are two broad ways to seed RNGs. The first is to seed them to behave randomly, while the other is to seed them to behave deterministically.

14.3.1 Random Seeding

The default behavior of random variables in *ns-3* is to automatically be seeded in such a way that the results of a given simulation vary from run to run. This behavior is good for quick testing of validity of code, but suffers from the lack of repeatability. The two methods for random seeding are outlined below.

Time of Day Seeding

The default behavior of RNG seeding in *ns-3* is to generate a seed from the current time of day, a common way to "randomly" seed RNGs.

/dev/random Seeding

Suppose the user is unhappy with the amount of true randomness present in the time of day method. He or she then has the option to specify a different method of random seeding via the system dependent /dev/random hardware device found on POSIX complaint and Unix-like operating systems. This method literally reads the seed values directly from /dev/random, and assures that the seed is from a true random source. A call to `RandomVariable::UseDevRandom()` preceding the first declaration of an RNG suffices to set up the generator to behave in this way.

14.3.2 Deterministic Seeding

The above random seeding cases function very well for general, quick tests of code coherency, but unfortunatly lack repeatability. Thus, the other way in which to use *ns-3* RNGs is in a deterministic fashion. This is done by setting a

global seed which doesn't change between runs. This effectively "locks-in" the RNGs to have the same behavior run to run, making the results predictable and useful for debugging purposes. This takes the form of a call of something like: `RandomVariable::UseGlobalSeed(ConstantSeed(1,2,3,4,5,6)).`

Incrementing the Run Number

A percieved downside of deterministic seeding is that in order to get results that ever change(for example, to collect statistical data about a simulation), one has to change the global seed between runs. In order to prevent this, the RNG packages support a specified run number that effectively steps all the RNGs to a different set of deterministic seeds. In this way, the same simulation can be run multiple times, each time with an incremented run number, and each time will give different results, but with consistency within a given run number. For example calling: `RandomVariable::SetRunNumber(27)` would increment the run to 27, giving generators that would be independent from say run number 26. Note that this code must be called before any declaration of an RNG.

Chapter 15

Command-line arguments

Simulations frequently need to have parameters that aren't fixed at compile time, but are rather tuned or modified just prior to execution. To aid in this facility *ns-3* will have a full featured set of tools to aid in command line argument passing and processing. The package of argument tools should be able to do things such as take user defined argument names to expect on the command line, and then populate specified variables from these command line values.

15.1 Usage

In particular, it will support integers, doubles, strings, and boolean command line arguments with equals signs, as well as flag type boolean values without equals signs. Assuming the user has configured the application correctly, the following type of simulation call will be supported: `./simulation run=20 linkspeed=56.6Kbps -enable-debug-output`

15.2 Automatically Generated Help

The argument processing capabilities will be able to automatically generate help information for calls to the simulation called wth arguments like –help or -help. This comes about from the user being able to specify help strings for each argument as he or she adds it to the processor. Suppose we have the following:

```
int main(int argc,char** argv)
{
  uint32_t nRuns;
  std::string linkSpeed;
  ArgProcessor::Add("runs",nRuns,3,"number of iterations to perform");
  ArgProcessor::Add("speed",linkSpeed, "10Mb/s", "default link speed");
  ArgProcessor::Process(argc,argv);
}
/* //output from the above
  ./foo --help
  --runs=[uint32_t:3]        : number of simulation iterations to perform.
  --speed=[string:10Mb/s]    : default link speed
  --help                     : print this help
*/
```

Chapter 16

Data Rates

When speaking of networking, data transmission speed is an important factor in specifying networks. These speeds are of course most commonly calculated as bit rates, that is number of bits per second. *ns-3* contains a class `DataRate` that can be used to represent data rates in simulations. The primary function of this class is to allow simple construction of data rates in terms of everyday units used, such as "Mbps" or "GB/s".

The `DataRate` class has a constructor that accepts a string which contains the required rate. There are three parts to any of these strings: the SI prefix, the units, and the time units. The following are the SI prefixes supported:

- 10^0 prefix - none
- 10^3 prefix - k
- 10^6 prefix - M
- 10^9 prefix - G

The units can be specified as bits (b) or bytes (B). Note this is an eight bit byte. Finally, the time units can be specified as either "ps" or "/s" to represent "per second". Example:

```
DataRate usb_low("1.5Mbps");
DataRate usb_full("1.5MB/s");
DataRate usb_hi("0.48Gbps");
DataRate legacy_network("1b/s");
```

The class also supports getting the bitrate as an integer number of bits per second with a call to `GetBitRate()`, and calculating the transmission time for a specified number of bytes. For example:

```
DataRate linkSpeed("3Mbps");
uint32_t packetSize = ... ; //some number of bytes; the size of a packet
double latency = linkSpeed.CalculateTxTime(packetSize);
```

Chapter 17

Debugging

17.1 Execution Tracing

From the perspective of a user of a complex system, it can be quite frustrating to write code that disappears into the system's black box only to return as an error. In some cases the underlying problem is caused by an unexpected interaction with another component deep within the system. Finding such problems is often realistically not possible for a user without a deeper understanding of what is happening inside the system. The ability to trace execution of certain components of a system are immensely useful in these circumstances; and allow more sophisticated users to understand nuances of the system and find workarounds, or to more fully specify reproducibility conditions for bug reporting.

From the perspective of a developer of such a complex system, it is often very useful to be able to get a quick idea of what is happening in the system, during development, testing and maintenance.

ns-3 addresses these issues by providing support for debug tracing functions that can be used in its constituent objects.

The requirements for the debug trace support were identified as:

- Code for a debug trace facility should be optimized out of production code. There should be no performance impact on working, production code;
- The debug trace facility should certainly be present in debug code (dbg-static or dbg-shared), possibly in otherwise optimized code enabled by compilation flag;
- The operation of the facility should be easily configured – that is one should not have to recompile to turn logging on and off for a particular problem;
- Tracing should be enabled certainly on a per-class basis. Tracing on an object-by-object basis may be desirable but will be harder to implement and configure;
- The facility should have selectable levels of verbosity, and default to a quiet mode in which nothing is output;
- Debug trace calls should be compatible with C++ `ostream`;

17.2 Debug design overview

The debug trace functionality is split into two main sections: configuration and tracing. Configuration is concerned with mechanisms for controlling the amount of information that is presented during the execution of the code; and tracing with the content of the information.

17.2.1 Configuration

The simplest form of configuration is through a static global variable. Each *ns-3* class for which tracing is enabled has a global variable indicating the selected verbosity level for the class. Typically the variable is set in a debugger or via method invocation.

The general idea is that as the selected verbosity level increases, the amount of information output increases. No explicit level definitions currently exist.

17.2.2 Tracing

The basic tracing functionality is provided through two macros:

- `NS3_TRACEALL`: Always prints a message if tracing is globally enabled;
- `NS3_TRACE`: Prints a message based on debug level configuration.

These macros are quite simple

```
#ifndef NDEBUG
#define NS3_DEBUG_ENABLE
#endif

#ifdef NS3_DEBUG_ENABLE
#define NS3_DEBUG(x) x
#else
#define NS3_DEBUG(x)
#endif

#define NS3_TRACEALL(traceout) \
  NS3_DEBUG(std::cerr << traceout << std::endl;)

#define NS3_TRACE(boolLevel, traceout) \
  NS3_DEBUG( \
    if (boolLevel) { \
      std::cerr << traceout << std::endl; \
    } \
  )
```

17.3 Using NS3_TRACEALL

The NS3_TRACEALL macro is used to print information irrespective of configuration. This can be extremely important debugging information that should always be seen by a user, or more mundanely things like banners or progress and status information in test programs.

```
int main (int argc, char *argv[])
{
  NS3_TRACEALL("Special Device Test")
  MacAddress addra("00:00:00:00:00:01");
  SerialNetDevice neta(a, addra);
  NS3_TRACEALL("neta.GetMtu() <= " << neta.GetMtu())
}
```

Note that NS3_TRACEALL is a preprocessor macro and so a semicolon at the end is not required. It doesn't hurt either.

Another macro, NS3_DEBUG, is available if one needs to add code to support the trace. For example, in the code snipped below, we declared a MacAddress to receive an address that will only be traced.

```
NS3_DEBUG (MacAddress addr = neta.GetAddress();)
NS3_TRACEALL("neta.GetAddress() <= " << addr)
```

Any code declared inside the NS3_DEBUG macro will be optimized out along with the actual trace code in production releases.

17.4 Using NS3_TRACE

The NS3_TRACE macro is used to print information under the control of of the trace configuration. This is how the various debug levels are implemented. In the code below, the first trace is output if the boolean expression gDebug evaluates to true. This is the case whenever the configured debug level is non-zero. The second trace is only output if the boolean expression gDebug > 1 evaluates to true.

```
  bool
Queue::Enque (const Packet& p)
{
  NS3_TRACE(qDebug,
    "Queue::Enque (" << &p << ")")

  NS3_TRACE(qDebug > 1,
    "Queue::Enque (): m_traceEnque (" << &p << ")")

  m_traceEnque (p);

  return RealEnque (p);
}
```

It is easy to see now that as the value of the trace configuration variable `qDebug` is increased, more debugging information will be printed.

Just like `NS3_TRACEALL`, `NS3_TRACE` is a preprocessor macro and so a semicolon at the end is optional.

Chapter 18

Acknowledgments

Principal Investigators:

- Sally Floyd (Berkeley ACIR)
- Tom Henderson (University of Washington)
- George Riley (Georgia Institute of Technology)
- Sumit Roy (Georgia Institute of Technology)

Developers:

- Raj Bhattacharjea (Georgia Institute of Technology)
- Craig Dowell (University of Washington)
- Mathieu Lacage (INRIA, France)

Contributors:

- Gustavo Carneiro
- Michelle Weigle (Old Dominion University)

Mathieu Lacage and George Riley have been instrumental in setting the early design framework for *ns-3*. Gustavo Carneiro has been very helpful as a bleeding-edge user and patch contributor.

This work is funded in part by the United States National Science Foundation (NSF) under contract numbers CNS-0551378, CNS-0551686, and CNS-0551706. All opinions expressed herein are the opinions of the authors and do not necessarily reflect those of the National Science Foundation.

Part VI

Appendices

Appendix A

Build system

This chapter provides an overview of the software build system.

A.1 Source code organization

Figure A.1 provides an overview of the *ns-3* source code organization. Section A.2 below details the build environment and options. The ns-3 library is split across multiple modules:

- core: located in src/core and contains a number of facilities which do not depend on any other module. Some of these facilities are OS-dependent.
- simulator: located in src/simulator and contains event scheduling facilities.
- common: located in src/common and contains facilities specific to network simulations but shared by pretty much every model of a network component.

A number of files exist in the top-level directory (SConstruct, build.py, and build.pyc) to coordinate the build process.

A.2 Build environment

We are trying the SCons build environment, and will fall back to the GNU build environment (autoconf, automake, libtool, gcc) if SCons doesn't work out. The default Windows build environment is still under consideration; probably Cygwin or mingw. Microsoft Visual C++ may be considered, but we are working through dll export/import issues.

Other compilers and build environments are outside the scope of the project, but we welcome anyone who wants to try alternatives to document how to use them.

See also the "BUILD" file in the top-level *ns-3* directory. Mathieu Lacage organized the *ns-3* build process and wrote the configuration files. The below is an expansion of the BUILD file.

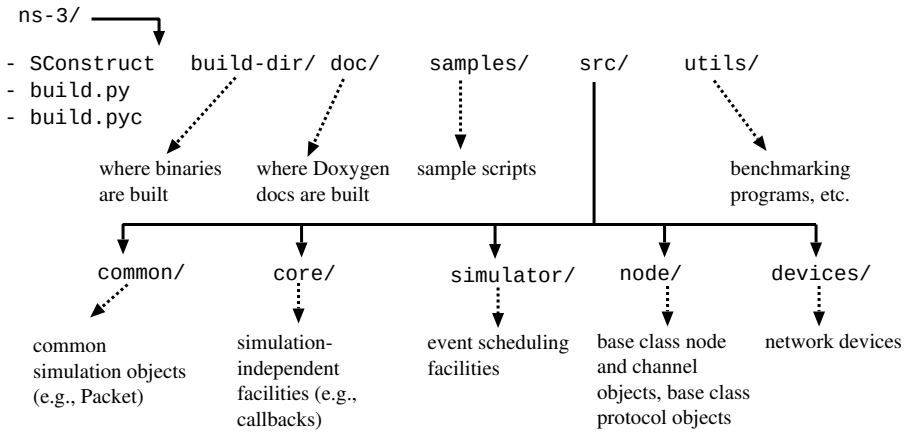

Figure A.1: Code organization for *ns-3* project.

A.2.1 SCons overview

From the SCons documentation:[1]

"SCons is a software construction tool (build tool, or make tool) implemented in Python, which uses Python scripts as "configuration files" for software builds. Distinctive features of SCons include: a modular design that lends itself to being embedded in other applications; a global view of all dependencies in the source tree; an improved model for parallel (-j) builds; automatic scanning of files for dependencies; use of MD5 signatures for deciding whether a file is up-to-date; use of traditional file timestamps instead of MD5 signatures available as an option; use of Python functions or objects to build target files; easy user extensibility."

If you want to build *ns-3*, you need to install SCons (see http://www.scons.org). SCons takes care of building the whole source tree using your system compiler. SCons 0.91.1 and 0.91.96 have been tested and are known to work on Linux FC5, Mac OS X and MinGW. OS X users may need to install from the DarwinPorts site.

SCons is configured by a "SConstruct" file in the top-level *ns-3* directory, as well as a "build.py" file.

To start a build, you can just type 'scons' which will generate a debug shared build by default, located in the directory 'build-dir/dbg-shared/bin' and 'build-dir/dbg-shared/lib'.

All builds are built with debugging symbols. Debugging builds enable asserts while optimized builds disable them. On platforms which support it, rpath is used which means that the executable binaries generated link explicitely against the right libraries. This saves you the pain of having to setup environment variables to point to the right libraries.

A.2.2 Options

- **verbose**: if you have installed SCons 0.91.96 or higher, the default build output is terse. To get a more verbose output, you need to set the 'verbose' variable to 'y'.
  Example: `scons verbose=y`
- **cflags**: flags for the C compiler.
  Example: `scons cflags="-O3 -ffast-math"`
- **cxxflags**: flags for the C++ compiler.

[1] (http://www.scons.org, Copyright 2001, 2002 by Steven Knight

Example: scons cxxflags="-O3 -ffast-math"

- **ldflags**: flags for the linker:
  Example: scons ldflags="-L/foo -L/bar"

Compilation flags can also be set in the `build.py` file. By default, the following are used:

```
if cc == 'gcc' and cxx == 'g++':
        common_flags = ['-g3', '-Wall', '-Werror']
        debug_flags = []
        opti_flags = ['-O3']
```

A.2.3 Build targets

- **doc**: build the doxygen documentation.
  Example: `scons doc`
- **dbg-shared**: a debug build using shared libraries. The files are built in `build-dir/dbg-shared/`.
  Example: `scons dbg-shared`
- **dbg-static**: a debug build using static libraries The files are built in `build-dir/dbg-static/`.
  Example: scons dbg-static
- **opt-shared**: an optimized build using shared libraries. The files are built in `build-dir/opt-shared/`.
  Example: `scons opt-shared`
- **opt-static**: an optimized build using static libraries. The files are built in `build-dir/opt-static/`.
  Example: `scons opt-static`
- **dbg**: an alias for dbg-shared
  Example: `scons dbg`
- **opt**: an alias for opt-shared
  Example: `scons opt`
- **all**: alias for dbg-shared, dbg-static, opt-shared and opt-static
  Example: `scons all`
- **gcov**: code coverage analysis. Build a debugging version of the code for code coverage analysis in `build-dir/gcov`. Once the code has been built, you can run various applications to exercise the code paths. To generate an html report from the gcov data, use the lcov-report target
  Example: `scons gcov`
- **lcov-report**: generate html report of gcov data. The output is stored in `build-dir/lcov-report/`.
  Example: `scons lcov-report`
- **dist**: generate a release tarball and zipfile from the source tree. The tarball and zipfile name are generated according to the version number stored in the SConstruct file.
  Example in SConstruct:

  ```
  ns3 = Ns3 ()
  ns3.name = 'foo'
  ns3.version = '0.0.10'
  ```

  Example command: `scons dist`
  Example output files:

```
foo-0.0.10.tar.gz
foo-0.0.10.zip
```

- **distcheck**: generate a release tarball and zipfile and attempt to run the 'all' target for the release tarball.
  Example: `scons distcheck`

A.2.4 How the build system works

The current build system defines what are called "ns3 modules": each module is a set of source files, normal header files and installable header files. Each module also depends on a set of other modules. We build modules automatically in the correct order. That is, we always start from the module which does not depend on any other module (core) and proceed with the other modules and make sure that when a module is built, all the modules it depends upon have already been built.

To build a module, we:

1. generate the .o files
2. link the .o files together
3. install the installable headers in the common directory `top_build_dir/include/ns3`.

This means that if you want to use a header from your own module, you should just include it: `#include "foo.h"` but if you want to include a header from another module, you need to include it with `#include "ns3/bar.h"`. This allows you to make sure that our "public" ns3 headers do not conflict with existing system-level headers. For instance, if you were to define a header called `queue.h`, you would include `ns3/queue.h` rather than `queue.h` when including from a separate module.

A.2.5 How to add files to an existing module

In the main SConstruct file, you can add source code to the `add_sources` method. For example, to add a foo.cc file to the core module, we coud do this: `core.add_sources ('foo.cc')`. Of course, if this file implements public API, its header should be installable: `core.add_inst_headers ('foo.h')`.

A.2.6 How to create a new module

First, create a new module in the top-level SConstruct file.

```
my_module = Ns3Module ('my', 'src/my_dir')
```

where the first argument is the name of the new module, and second argument is the directory in which all source files for this module reside. Next, add it to the build system:

```
ns3.add (my_module)
```

Next, specify module dependencies; for example, if it depends on the ipv4 and core modules, add:

```
my_module.add_deps (['core', 'ipv4'])
```

Next, add source code to this module:

```
my_module.add_sources ([
        'my_a.cc',
        'my_b.cc',
        'my_c.cc'
])
my_module.add_sources ([
        'my_d.cc'
])
```

To add headers which are not public, do

```
my_module.add_headers ([
        'my_a.h',
        'my_c.h'
])
```

To add headers which are public:

```
# add headers which are public
my_module.add_inst_headers ([
        'my_b.h'
])
my_module.add_inst_headers ([
        'my_d.h'
])
```

If you need to link against an external library, you must add 'external' dependencies. Here as an example, the pthread library:

```
my_module.add_external_dep ('pthread')
```

Finally, note that by default, a module is conceptually a library. If you want to generate an executable from a module you need to:

```
my_module.set_executable ()
```

A.2.7 Build output

Targets end up in the `build-dir/` directory. Different build targets end up in different directories (e.g., `dbg-shared/`).

ns-3 build process builds a library for each module in the `src/` directory, such as `libcommon`, `libcore`, and `libsimulator`. These are found in the `lib/` directory in the build directory. Executables are found in the `bin/` directory; these executables are linked against the module libraries during the build process.

A.2.8 Code coverage

On x86 systems when using gcc, one can use gcov and lcov to profile the code coverage of a set of tests.

```
scons gcov
(execute some of the executables in the gcov/bin directory)
scons lcov-report
```

The resulting html will end up in the `build-dir/lcov-report/index.html`.