ns-3 Software Architecture

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

July 15, 2007

Introduction

This *ns-3* design document is one of a set of project documents:

- Software Architecture (this document)
- User's Guide (not yet written)
- Contributor's Guide (not yet written)

This document is written in Latex and is to be maintained in revision control on the *ns-3* code server. Both PDF and HTML versions should be available on the server. Changes to the document should be discussed on the ns-developers@isi.edu mailing list.

Contents

1 Introduction **3**
- 1.1 *ns-3* Overview . . . 3
- 1.2 Longer-term vision . . . 3
- 1.3 Outline . . . 4

2 Software Architecture **5**
- 2.1 Basics . . . 5
- 2.2 Use cases . . . 5
- 2.3 Class and object design . . . 7
  - 2.3.1 Component system . . . 7
  - 2.3.2 Object creation . . . 8
- 2.4 Memory Management . . . 9
  - 2.4.1 Reference counting smart pointer . . . 10
- 2.5 Configuration . . . 10
  - 2.5.1 Stock topology code . . . 10
  - 2.5.2 Default values and command line arguments . . . 10
- 2.6 Tracing . . . 11
- 2.7 Scaling . . . 12
- 2.8 Emulation . . . 12
- 2.9 Scripting . . . 12

3 Key simulation objects **13**
- 3.1 Node . . . 13
- 3.2 NetDevice and Channel . . . 14
- 3.3 Packet . . . 15
- 3.4 Sockets and Applications . . . 15
  - 3.4.1 Sockets . . . 16

4 Core Modules **17**
- 4.1 Event Scheduling . . . 17
  - 4.1.1 Simulation time . . . 17
  - 4.1.2 Event creation and expiration . . . 17
- 4.2 Callbacks . . . 18

5 Internet Node **19**
- 5.1 InternetNode members . . . 19
- 5.2 Send packet processing chain . . . 20
- 5.3 Receive packet processing chain . . . 20

6 ns-3 routing **22**
6.1 Overview 22
6.2 RouteManager 23
6.3 Node routing interface 24
6.4 Proposed roadmap 26

1 Introduction

This document provides an overview of the high-level goals and software architecture for the *ns-3* network simulator. *ns-3* is aimed at eventually replacing the *ns-2* simulator. This document is intended to provide a brief architectural overview of *ns-3*, to complement reading the code and main source code documentation, which is in Doxygen[1] format.

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-based 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, but the goal is for full or near-full API support at the Python level.

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

- Modularity of components
- Scalability of simulations
- Integration/reuse of externally developed code and software utilities
- 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 we are studying approaches to allow *ns-2* to be run as part of *ns-3*, as well as studying which models will be ported from *ns-2* to work natively in *ns-3*.

1.2 Longer-term vision

The PIs and developers on the project envision that *ns-3* can become more than a basic iteration of previous simulator approaches. 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), encapsulation, 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. Abstraction layers, interfaces, and new techniques for supporting implementation code within the *ns-3* environment, such as ports of popular operating system implementations and routing daemons;

[1] http://www.nsnam.org/doxygen/index.html

2. Support for standard input and output file formats, so that existing tools can be used for generating simulation input and analyzing simulation output (e.g., pcap-formatted traces for viewing with tcpdump);
3. Techniques to allow users to easily migrate experiments 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.

1.3 Outline

This document is organized as follows:

- Chapter 2 describes the overall end-to-end software architectural model
- Chapter 3 introduces the key objects in the system relating to sending and receiving packets: nodes, network devices, channels, packets, and sockets.
- Chapter 4 describes core objects in the simulator.
- Chapter 5 outlines how the InternetNode object sends and receives packets.
- Chapter 6 includes a proposal for how to add routing to *ns-3*.

2 Software Architecture

This chapter provides an introductory software architectural overview of *ns-3*, including use cases, architecture for reusable components, design for configuration, memory management policy, and strategy for integrating outside and legacy code.

2.1 Basics

ns-3 is a user-space program that runs on Unix- and Linux-based systems and on Windows (currently via Cygwin and possibly via native win32 APIs in the future). It is written in C++, with a planned Python scripting interface(s) for users. The focus is on IPv4 and IPv6-based networks, but other non-IP architectures such as sensors or DTNs are to be supported. *ns-3* is meant to be modifiable and extendable by users; some users will be able to use example scripts that are provided, but it is expected that most (research) users will want to either write new scripts or modify or add to the simulator models in some way. Source code distributions are therefore expected to be the preferred means for distributing *ns-3*.

ns-3 contains support for the following:

- construction of virtual networks (nodes, channels, applications) and support for items such as event schedulers, topology generators, timers, random variables, and other objects 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 has a modular implementation containing a `core` library supporting generic aspects of the simulator (debugging objects, random number generators, smart pointers, callbacks, unit tests, reference list), and a `simulator` library defining simulation time objects, schedulers, and events. A `common` library defines objects that are independent of specific network architectures, such as generic packets and tracing objects. Finally, the `node` library defines abstract base classes for fundamental base objects in the simulator, such as nodes, channels, and network devices. Internet-related models (IP and transport protocols) are found in the `internet-node` library. Specific devices such as Ethernet are in `device` libraries. Users may write and link their own libraries. The modular implementation allows for smaller compilation units. *ns-3* executable programs may be built to either statically or dynamically link the libraries.

2.2 Use cases

To introduce the design of *ns-3* we first review design issues and usage models that have arisen with *ns-2*, and mention trends in simulation use within the networking research community.

- **Model extensibility.** Most research users want to extend the simulator by writing new simulation scripts, modifying existing models, or writing new models. To facilitate model modification, *ns-3* continues the use of object-oriented design with polymorphic classes, allowing users to subclass the aspects that they wish to modify. To facilitate the addition of new models, *ns-3* adopts a component-based architecture for compile-time or run-time addition of new models, interface aggregation, and encapsulation, without requiring modification of the base models of *ns-3*.

- **Simulation code reuse.** Many users start their work with *ns-2* by adapting existing code. Some common code is written in terms of base-class object pointers, allowing for run-time substitution of subclassed objects. *ns-3* will use several techniques to facilitate simulation code reuse, such as inheritance to extend existing classes, the provision of (extensible) stock topology objects, simulation frameworks that are easily modifiable, an example script repository, and a system for run-time configuration of classes and default values.
- **Run-time configuration.** *ns-3* provides a flexible technique to allow users to redefine default values and class types without recompiling the simulator. The default value database is integrated with a command-line argument parsing facility, making all the variables configurable from the command-line as well.
- **Tracing.** *ns-3* features a callback-based approach to tracing that decouples tracing sources from tracing sinks and that is focused on flexibility for the user. Packet traces will be made available in libpcap format, to allow for post-processing tools built around that trace format. Built-in statistics will also be widely available.
- **Scaling.** *ns-3* will include techniques for improving the scalability of simulations, including distributed simulation techniques introduced with PDNS and GTNetS, scalability techniques introduced for wireless simulations such as caching of computationally-intensive results, and flexibility in tracing infrastructure (to avoid large traces).
- **Software integration.** *ns-3* is oriented towards the reuse of existing software such as routing daemons, applications, and kernel code. The design is built around encapsulation techniques that decouple the interface from the implementation, an architecture that better mirrors how real-world devices are built (e.g., explicitly handling multiple interfaces per node), and an abstraction library that allows implementation code to run in both real and simulated environments.
- **Network emulation.** Increasingly, network research that involves simulation also includes an experimental component, with facilities such as PlanetLab, Emulab, and ORBIT. Researchers would like to more easily move between simulation and experimental domains. The *ns-3* design is intended to facilitate this interaction between simulation and experiments, with a packet design oriented towards serialization and deserialization, and encapsulation techniques that will allow real application and kernel code to run in the simulator, thereby improving traceability to real-world implementations.
- **Scripting** The primary *ns-3* user interface at present is a C++ "main" program, and we expect that C++ will continue to be a preferred language for many users. However, *ns-3* will also feature Python bindings allowing for users to define scripts and replaceable components in Python.

We organize the rest of the discussion in this chapter as follows:

1. Class and object design
2. Memory management
3. Configuration
4. Tracing
5. Scaling
6. Emulation
7. Scripting

The next chapter goes into more detail on the Node, Channel, and Packet object designs.

2.3 Class and object design

This section describes the C++ class design for *ns-3*objects. In brief, the design patterns in use include classic object-oriented design (polymorphic interfaces and implementations), separation of interface and implementation, the non-virtual public interface design pattern, object and interface aggregation, a type-safe query interface, a run-time replaceable components system, and reference counting for memory management. Those familiar with component models such as COM or Bonobo will recognize elements of the design in *ns-3*, although the *ns-3*design is not strictly in accordance with either.

2.3.1 Component system

The *ns-3* component system is motivated in strong part by a recognition that a common use case for *ns-2* has been the use of polymorphism to extend protocol models. For instance, specialized versions of TCP such as RenoTcpAgent derive from (and override functions from) class TcpAgent.

However, two problems that have arisen in the *ns-2* model are downcasts and "weak base class." Downcasting refers to the procedure of using a base class pointer to an object and querying it at run time to find out type information, used to explicitly cast the pointer to a subclass pointer so that the subclass API can be used. Weak base class refers to the problems that arise when a class cannot be effectively reused (derived from) because it lacks necessary functionality, leading the developer to have to modify the base class and causing proliferation of base class API calls, some of which may not be semantically correct for all subclasses.

ns-3 is using a version of the query interface design pattern to avoid these problems. This design is based on elements of the Component Object Model design[1] and GNOME Bonobo,[2] although full binary-level compatibility of replaceable components is not supported and we have tried to simplify the syntax and impact on model developers. The aspects of COM that we are using provide:

- a component-oriented programming model, based on separation of interface and implementation. Interface objects are what client code uses to talk to the underlying implementation. When the class design follows this pattern, it allows components supporting similar interfaces to be swapped out.
- what if interfaces of replaceable components are not the same? COM provides a QueryInterface capability which, in our implementation, provides a type-safe way to query whether an object has a given capability or interface. The key to this architecture is that interfaces can be added or aggregated at run-time to objects without requiring rebuilding of the base classes, thereby avoiding weak base classes and the need for client-side C++ downcasts to provide run-time type information (RTTI).
- a system of unique identifiers for interfaces and classes.
- a component manager that is able to instantiate factories and objects themselves based on the identifiers mentioned above.
- a memory management policy rooted in reference counting.

We do not enforce the COM rule that interfaces are pure abstract and that one must separate the interface from implementation. A different, fuller port of COM to *ns-3* was prototyped by Craig Dowell,[3] who initially suggested the use of COM concepts and implementation for *ns-3*.

[1] http://en.wikipedia.org/wiki/Component_Object_Model
[2] http://en.wikipedia.org/wiki/Bonobo_%28computing%29
[3] http://code.nsnam.org/craigdo/ns-3-com

Query interface example

Query interface is a type-safe way to achieve a safe downcasting and to allow interfaces to be aggregated to an object. Objects using the query interface must inherit from the Interface base class.

An example of the use of query interface is shown below. Consider a node pointer `n0` that points to an InternetNode object with an implementation of IPv4. The client code wishes to configure a default route. To do so, it must access an object within the node that has an interface to the IP forwarding configuration. It performs the following two steps:

```
Ptr<IIpv4> ipv4 = n0->QueryInterface<IIpv4> (IIpv4::iid);
ipv4->SetDefaultRoute (Ipv4Address ("10.1.1.2"), 1);
```

In the first line a (smart) pointer of type IIpv4 ("interface to IPv4") is declared and assigned to the result of a QueryInterface on the node for the interface type IIpv4. This pointer value will be returned null if the node doesn't support the requested interface. If non-null, this pointer can be used like a traditional pointer to access the API of the IIpv4 object.

To summarize, two benefits that we expect to leverage from this are as follows:

- **Encapsulation:** By separating interface from implementation, it permits implementors to replace elements of the protocol stack while remaining compliant with client code that supports the same interface. For example, one type of node may include native *ns-3* models of protocols, while another may be a port of a Linux stack, and both may be accessed by the same interface.
- **Aggregation:** QueryInterface allows for aggregation of interfaces at run time. For instance, an existing Node object may have an "Energy Model" object and its interface aggregated to it at run time (without modifying and recompiling the node class). An existing model (such as a wireless net device) can then query interface for the energy model and act appropriately if the interface has been either built in to the underlying Node object or aggregated to it at run time.

We hope that this mode of programming will require much less need for developers to modify the *ns-3* base classes or libraries.

See also the `samples/main-query-interface.cc` program.

2.3.2 Object creation

Objects in C++ may be statically, dynamically, or automatically created. This holds true for *ns-3* also, but some objects in the system– those using the replaceable component system– have some additional frameworks available. Specifically, reference counted objects are dynamically allocated using operator new, a templated MakeNewObject method, or an *ns-3* component manager.

The ComponentManager class is inspired by COM and is a class used to create any Interface class by ClassId, where ClassId is a symbolic name associated to a particular class. Each class using the component manager declares a unique `ns3::ClassId` static variable that is bound to a constructor. The following code shows how the component manager can be used to create new objects of type A:

```
Ptr<A> a = 0;
a = ComponentManager::Create<A> (A::cid, A::iid);
```

The above code (from the unit tests for component-manager.cc) creates a class A (which is subclassed from Interface) and returning a pointer to A (as specified by A's interface ID)..

The above code sample can be changed in a few ways. First, if A statically aggregates interface B, a pointer to interface B can be returned even if the underlying object is of type A:

```
Ptr<B> b = 0;
b = ComponentManager::Create<A> (A::cid, B::iid);
```

Finally, the system accommodates non-default constructors. Assume that another constructor for A exists that takes a boolean argument, such as `class A::A (bool bo)`. If the constructor for this class has registered a new class Id (such as cidOneBool), the following can be called:

```
Ptr<B> b = 0;
b = ComponentManager::Create<A,bool> (A::cidOneBool, B::iid, true);
```

where the last parameter is the passed-in boolean value to A's constructor, and again assigning returning the interface pointer B to the created object of type A. The classIds can be overridden at run time also by the default value system described below.

If a reference counted object is being new'ed and assigned to a reference counting smart pointer (class Ptr), then a templated helper function is available and recommended to be used:

```
ns3::Ptr<B> b = ns3::Create<B> ();
```

This is simply a wrapper around operator new that correctly handles the reference counting system.

2.4 Memory Management

Memory management in a C++ program is a complex process, and is often done incorrectly or inconsistently. We have settled on a reference counting design described as follows.

All objects using reference counting maintain an internal reference count to determine when an object can safely delete itself. Each time that a pointer is obtained to an interface, the object's reference count is incremented by calling `Ref()`. It is the obligation of the user of the pointer to explicitly `Unref()` the pointer when done. When the reference count falls to zero, the object is deleted.

- When the client code obtains a pointer from the object itself through object creation, or via QueryInterface, it does not have to increment the reference count.
- When client code obtains a pointer from another source (e.g., copying a pointer) it must call `Ref()` to increment the reference count.
- All users of the object pointer must call `Unref()` to release the reference.

The burden for calling `Unref()` is somewhat relieved by the use of the reference counting smart pointer class described below.

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

Packet objects are handled differently (without reference counting); their design is described in the next chapter.

2.4.1 Reference counting smart pointer

ns-3 provides a smart pointer class similar to `Boost::intrusive_ptr`. This smart-pointer class assumes that the underlying type provides a pair of Ref and Unref methods that are expected to increment and decrement the internal refcount of the object instance. We saw an example of this class in the query interface code above.

This implementation allows you to manipulate the smart pointer as if it was a normal pointer: you can compare it with zero, compare it against other pointers, assign zero to it, etc.

It is possible to extract the raw pointer from this smart pointer with the GetPointer and PeekPointer methods.

If you want to store a newed object into a smart pointer, we recommend you to use the MakeNewObject template functions to create the object and store it in a smart pointer to avoid memory leaks. These functions are really small conveniance functions and their goal is just is save you a small bit of typing.

2.5 Configuration

Configuration of objects is typically done by accessing an object's public API to change the values of member variables. That is no different in *ns-3* but the design tries to ease this for users with the following techniques.

2.5.1 Stock topology code

A number of static methods are being defined to aid in topology construction. These objects typically use base class pointers to refer to constituent objects (enabling software reuse) and are therefore a primary benefactor of the COM-like frameworks (QueryInterface, Component Manager) described above. For the moment, only a few PointToPointTopology objects are available (in `src/devices/p2p/p2p-topology.cc,h`) but more topologies such as WirelessGrid are planned.

For example, the following method constructs a point-to-point link (using PointToPointChannel and PointToPointNetDevice objects) between two nodes n1 and n2, with the specified dataRate and one way progagation delay. It essentially wraps a bunch of low-level API calls to create these NetDevices and Channel. The type of objects used in this topology can be overridden as long as they derive from the common base classes used in these topology objects. Users may write their own topology objects, but *ns-3* will maintain a few.

2.5.2 Default values and command line arguments

Simulation users often want to run many instances with slightly different parameters. *ns-2* had a system whereby users could change the value of a C++ variable if it was suitably bound (see the `tcl/lib/ns-default.tcl` script of *ns-2*).

In *ns-3*, we have developed the following system for default values, and have hooked it into a command-line argument parsing facility. The basic idea is to use a templated global variable facility to store bindings between string names of variables, "help" text on allowable parameters, and the default value itself. This avoids users needing to rebuild core libraries to change parameters, and allows users to avoid rebuilding any files at all if the command-line facility is used.

The program in `samples/main-default-value.cc` shows how this facility can be used. Briefly, any variable of a supported type in the system can be bound to a unique string by first declaring a static variable such as

```
static IntegerDefaultValue<int> defaultTestInt1 ("testInt1", "helpInt1", 33);
```

which declares that testInt1 is an integer with a default value of 33. The second parameter is a string that can be modified by the developer to encode whatever information is useful (e.g., units). Then, any actual integer in the system can be later assigned to the value of defaultTestInt1, as typically done in an object's constructor.

If a variable in the system has been bound to the string "testInt1", the following C++ statement (typically invoked near the top of a main program) will cause it to be initialized instead to e.g. the integer value 57:

```
Bind("testInt1", "57");
```

While a user can change this default by modifying the main program, the command line can be used as well. Running `"./sample-default-value -help"` will cause a list of possible configurable values to be printed out. For this example, the following string is printed:

```
--testInt1=[int32_t(-2147483648:2147483647):33] helpInt1
```

This tells the user that testInt1 is of type `int32_t` with a range of values specified between the parentheses, and a default value of 33 (that can be overridden).

This facility can also be used to swap out the type of an object at run-time, if the particular class has been integrated into the system. For instance, the file `examples/simple-p2p.cc` shows a line as follows:

```
Bind ("Queue", "DropTailQueue");
```

where DropTailQueue is a subclass of class Queue. This type of binding will allow callers of the `Queue::CreateDefault ()` factory method to obtain a suitably subclassed Queue object.

Consult the `samples/main-default-value.cc` example program for more information on how to use this facility.

2.6 Tracing

The design objective has been to offer the user a lot of flexibility in selecting which events to monitor, and to allow users freedom to use possibly complex logic to decide what things to log to trace files or to perform inline statistics calculations.

To provide this flexibility, every model must define a set of trace event sources. Each of these trace event source can generate one type of event and can specify any number of arguments to convey per-event information from the trace event source to the listening trace event sinks.

While this design allows users to hand-specify a different trace sink to each trace source, ns-3 also provides a set of simple trace helpers which perform bulk connection of the default trace sources to a set of trace sinks which generate trace files in various specific formats. For example, pcap output can be trivially generated for the default ipv4 stack by instantiating an object of type PcapTrace and calling its TraceAllIp method. The example file `examples/simple-p2p.cc` contains examples for producing both ascii and pcap traces using this high-level API.

To integrate in this framework, model developers need to:

- define and instantiate a set of trace sources of type CallbackTraceSource
- trigger trace events by invoking each of the trace source with the per-event arguments needed

- implement a method named CreateTraceResolver which takes a TraceContext as argument and returns a TraceResolver. Implementing this method is pretty trivial: it is a matter of instanciating a CompositeTraceResolver and register each trace source in it.

Later, when the time comes to connect the user's trace sinks (that is, the user's callbacks. See section 4.2) to the model's trace sources, the user can use the TraceRoot::Connect method which takes as an argument a string pattern which identifies the set of trace sources stored in trace resolver instances to connect. For example, a string pattern could look like '/nodes/*/netdevices/*/*' which would identify all trace events in all netdevice objects contained in all nodes.

2.7 Scaling

Note: the ns-3.0.3 release does not include specific support for scaling techniques. This section will be added at a later date.

2.8 Emulation

Note: the ns-3.0.3 release does not include support for emulation.

2.9 Scripting

Gustavo Carneiro and Craig Dowell are working on Python scripting; check the ns-developers list for discussion of the design.

Note: the ns-3.0.3 release does not include support for Python scripting. This section will be added at a later date.

3 Key simulation objects

This chapter walks through the primary simulation objects in the simulator, relating to the sending and receiving of packets between nodes. Figure 3.1 depicts, at a high-level, the objects we will discuss in this chapter: Node, NetDevice, Channel, Packet, and interface aspects thereof.

3.1 Node

A Node is an abstract base class in *ns-3*. It contains only a few objects: a unique integer ID, a system ID (for distributed simulation), a list of NetDevices, and a list of Applications. Figure 3.1 depicts this high-level view.

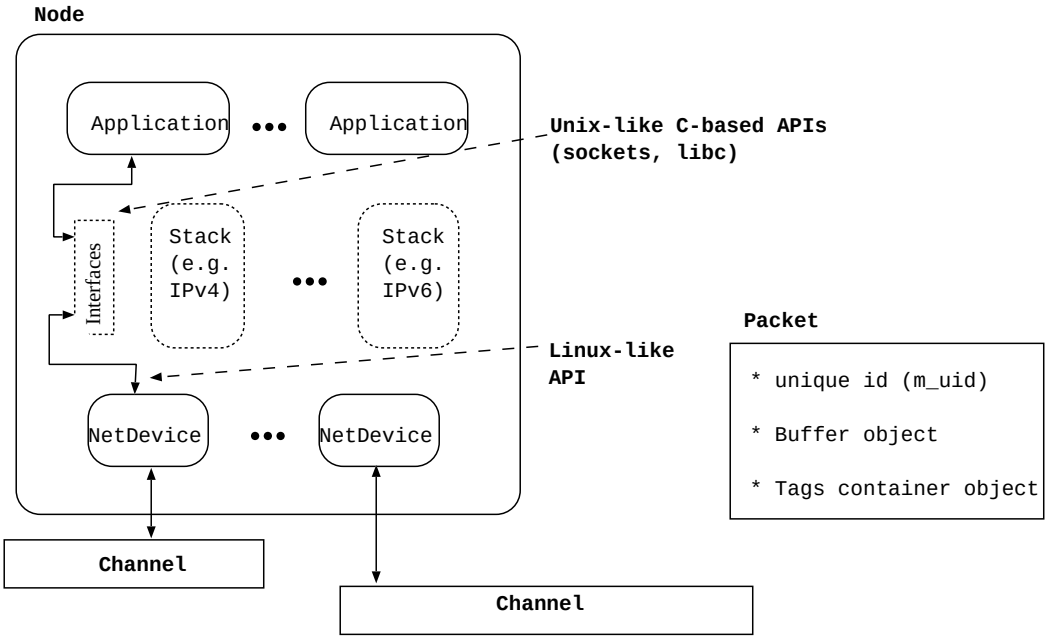

Figure 3.1: High-level node architecture.

Users can create their own Node subclasses, and *ns-3* will provide a few. Currently, `class InternetNode` is provided, which implements a rudimentary UDP/IPv4 stack.

The design tries to avoid putting too many dependencies on the base class Node, Application, or NetDevice for the following:

- IP version, or whether IP is at all even used in the Node.
- implementation details of the IP stack

The design therefore uses the design pattern of software encapsulation to allow Applications and NetDevices to talk to implementation- independent interfaces (that can be queried via QueryInterface– see section 2.3) of the underlying TCP/IP implementations.

For instance, we expect to support a native *ns-3* version of TCP/IP as well as ported Linux or FreeBSD stacks. These implementation details can be hidden behind an IPv4 interface object that is queried by the application or scenario developer. If users want to experiment with non-IP stacks, they can do so without having IP dependencies on the NetDevices, Channels,

and Applications. This is why the Stack objects in Figure 3.1 are illustrated with dotted lines; these may be built quite differently for different Node subclasses. We try to provide an interface to the NetDevice corresponding to the device-independent sublayer in Linux, and model the interface on the top of the stacks using typical Unix-like abstractions found in (C-based) sockets API and other system calls such as found in libc or other utilities.

3.2 NetDevice and Channel

A key node object is `class NetDevice`, which represents a physical interface on a node (such as an Ethernet interface). We discuss also in this section the `class Channel`, which is closely coupled to the attached NetDevices.

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 3.2). The top interface of NetDevice approximates the point in the Linux kernel where `dev_queue_xmit()` is called. The data members of NetDevice are similar to those found in Linux `struct net_device`. The IPv4 or IPv6 portion of a device (`struct in_device`) is modeled by a separate object on top of NetDevice (not discussed in this section).

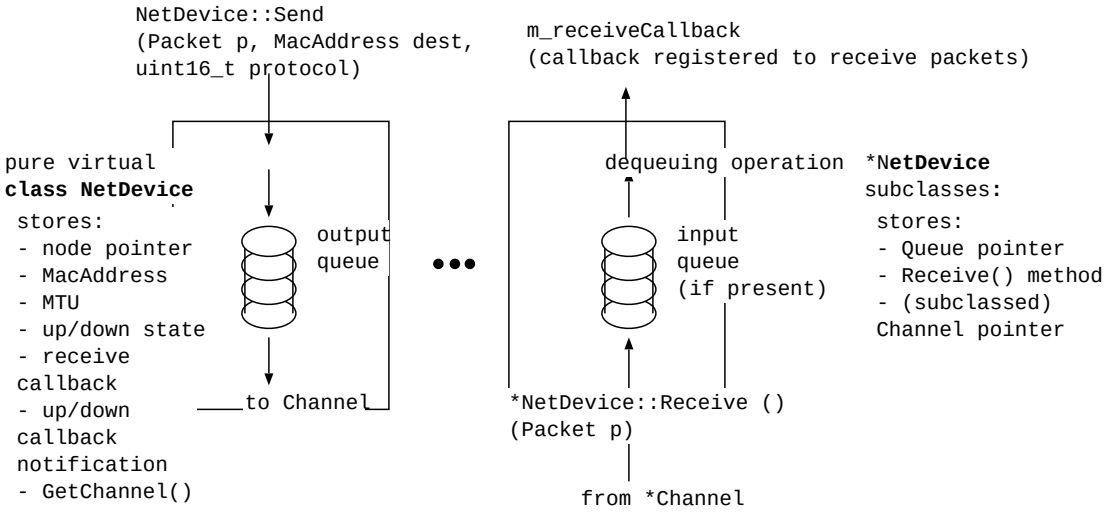

Figure 3.2: Overview of boundary between Network Device and upper layer (typically layer-3).

Figure 3.2 illustrates some of the main objects and actions involving sending a packet up and down the stack. First, there is an abstract `class NetDevice` that implements a Node pointer, MacAddress, string name (e.g., "eth0"), MTU, and has a flag for setting the state to be up or down. Two callbacks are included; the first allows a higher-layer protocol to register a function to be used to send the packet up the stack; this callback is present to decouple the NetDevice from the higher layer protocol above (typically layer-3 but may also be something like a bridging layer), as described in the previous section. Another callback allows the NetDevice to notify listeners of a change in state. Finally, there is a method provided to return a base class Channel pointer, which is forwarded to a NetDevice subclass that actually has the pointer.

NetDevices in use in the simulation will all subclass from this base class; an example is in `src/devices/p2p-net-device.cc,h`. These subclasses are matched to a particular corresponding channel type. That is, for example, a PointToPointNetDevice is attached to a PointToPointChannel. This convention provides type-safety in avoiding the connection of incompatible Channel and NetDevice types. The subclass (denoted `*NetDevice` in the figure) also provides a Receive() method to allow packets to be sent to it from the Channel; e.g. PointToPointChannel calls PointToPointNetDevice::Receive(). Any queue implementations are stored in these subclasses.

Packets traversing the stack in the outbound direction call the base class NetDevice::Send() which forwards the packet to

the appropriate subclass method. Packets traversing the stack in the inbound direction will call the callback registered with m_receiveCallback when the NetDevice is done processing the packet and wants to hand it to the higher layer.

3.3 Packet

The design of the Packet framework of *ns-3* was heavily guided by a few important use-cases:

- avoid changing the core of the simulator to introduce new types of packet headers or trailers
- maximize the ease of integration with real-world code and systems
- make it easy to support fragmentation, defragmentation, and, concatenation which are important, especially in wireless systems.
- make memory management of this object efficient
- allow actual application data or dummy application bytes for emulated applications

ns-3 Packet objects contain a buffer of bytes: protocol headers and trailers are serialized in this buffer of bytes using user-provided serialization and deserialization routines. The content of this byte buffer is expected to match bit-for-bit the content of a real packet on a real network implementing the protocol of interest.

Fragmentation and defragmentation are quite natural to implement within this context: since we have a buffer of real bytes, we can split it in multiple fragments and re-assemble these fragments. We expect that this choice will make it really easy to wrap our Packet data structure within Linux-style skb or BSD-style mbuf to integrate real-world kernel code in the simulator. We also expect that performing a real-time plug of the simulator to a real-world network will be easy.

Because we understand that simulation developers often wish to store in packet objects data which is not found in the real packets (such as timestamps or any kind of similar in-band data), the *ns-3* Packet class can also store extra per-packet "Tags" which are 16 bytes blobs of data. Any Packet can store any number of unique Tags, each of which is uniquely identified by its C++ type. These tags make it easy to attach per-model data to a packet without having to patch the main Packet class or Packet facilities.

Memory management of Packet objects is entirely automatic and extremely efficient: memory for the application-level payload can be modelized by a virtual buffer of zero-filled bytes for which memory is never allocated unless explicitely requested by the user or unless the packet is fragmented. Furthermore, copying, adding, and, removing headers or trailers to a packet has been optimized to be virtually free through a technique known as Copy On Write.

3.4 Sockets and Applications

Applications are user defined processes that generate traffic to send across the networks to be simulated. *ns-3* provides a framework for developing different types of applications that have different traffic patterns. There is an Application base class that allows one to define new traffic generation patterns via inheritance from this class. Then one simply creates the application and associates it with a node, and the application will send traffic down the protocol stack. The way that applications on a node communicate with the node's protocol stack is via sockets.

3.4.1 Sockets

The sockets API exported to *ns-3* attempts to mimic the standard BSD sockets API. The major difference in the implementation is that while BSD socket calls are synchronous (that is, they do not return control to their caller until they complete), the *ns-3* socket API calls return immediately. This is due to the fact that in a simulation environment where one machine is simulating possibly thousands of socket calls across different simulated machines simultaneously, the simulator simply cannot afford to wait for the socket function call to return. The way the software handles the situation instead is by returning immediately, then using callbacks when other portions of the code need to be notified of a socket event. For example, when in the course of the simulation a socket is directed to listen() on a specific port, the caller also provides a callback to handle when the socket receives a connection request. The listen() method returns immediately, and then whenever the socket receives the connection, it invokes the callback to handle the connection. Similar things happen for the other common socket APIs, like send(), connect(), and bind().

4 Core Modules

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.

4.1 Event Scheduling

The ns-3 event scheduling framework was designed with the following use-cases in mind:

- maximize code portability by ensuring reproducible time calculations in user models.
- make it possible to increase the precision of the internal time variable in the future.
- make it easy to associate a specific function to be called when a specific event expires
- make it easy to pass per-event data from the point when the event is scheduled to the point when the event expires

4.1.1 Simulation time

Simulation time is kept track of internally using a 64bit integer in units of nanoseconds. To make sure that this internal variable can be easily changed to represent a higher-precision time or that we can use a variable with a larger dynamic range, user programs never access directly this time variable. Instead, the current simulation time is exported to the user through a single method Simulator::Now () which returns an opaque object of type Time. Users can also easily create instances of this type through the functions Seconds, MilliSeconds, MicroSeconds, or NanoSeconds each of which takes a single argument in the specified unit and returns an instance of a Time object.

Instances of the class Time can be used just like normal integers or floating-point values: they support all the normal arithmetic operators and can be converted to values in a specific time unit with Time::GetSeconds, Time::GetMilliSeconds, Time::GetMicroSeconds, and, Time::GetNanoSeconds.

These instances of the class Time store their time value in a 64.64bit fixed-point integer variable. That is, the user-visible time variables are kept track of with 64 bits of fractional integer precision. If users are careful to perform all their arithemtic operations on Time variables, they can easily ensure that their code will behave exactly in the same way on multiple platforms.

4.1.2 Event creation and expiration

To schedule an event, users can call any of the Simulator::Schedule functions:

```
void MyEvent (double a)
{
  ...
}

EventId id = Simulator::Schedule (Seconds (10.0), &MyEvent, 3.1415);

NS_ASSERT (is.IsRunning ());
```

```
id.Cancel ();
NS_ASSERT (is.IsExpired ());
Simulator::Remove (id);

Simulator::Run ();
```

These Schedule functions all take as first argument a Time variable. Their second argument is always a function pointer and the other arguments are the values which will be passed to the user event function when the event expires. There can be up to 5 values to be passed to the user function.

Once scheduled, any event can be canceled (its cancel bit is set to true) or removed (it is removed from the event list): both operations will ensure that the event never expires.

4.2 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. Up to five arguments can be passed with the function pointer 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. A sample program is found in `samples/main-callback.cc`

5 Internet Node

Class InternetNode defines the canonical IP-based node in the simulator. Recall in Chapter 3 that class Node is an abstract base class that has a list of NetDevices and a list of Applications, but the protocol layers between the Applications and NetDevices are undefined in this base class. Class InternetNode provides an implementation of these IP-based layer-3 and layer-4 protocols. We envision that ports of other operating systems (such as Linux or FreeBSD) will be defined as other types of Node, hopefully with similar configuration interfaces.

This chapter provides a brief overview of the objects that make up the layer-3 and layer-4 plumbing, and by way of description, traces the path of a packet through these objects.

5.1 InternetNode members

The InternetNode::Construct() function (called by the object constructor) describes what makes up an InternetNode.

from internet-node.cc

```
54 void
55 InternetNode::Construct (void)
56 {
57   Ptr<Ipv4L3Protocol> ipv4 = Create<Ipv4L3Protocol> (this);
58   Ptr<ArpL3Protocol> arp = Create<ArpL3Protocol> (this);
59   Ptr<UdpL4Protocol> udp = Create<UdpL4Protocol> (this);
60
61   Ptr<L3Demux> l3Demux = Create<L3Demux> (this);
62   Ptr<Ipv4L4Demux> ipv4L4Demux = Create<Ipv4L4Demux> (this);
63
64   l3Demux->Insert (ipv4);
65   l3Demux->Insert (arp);
66   ipv4L4Demux->Insert (udp);
67
68   Ptr<UdpImpl> udpImpl = Create<UdpImpl> (udp);
69   Ptr<ArpPrivate> arpPrivate = Create<ArpPrivate> (arp);
70   Ptr<Ipv4Impl> ipv4Impl = Create<Ipv4Impl> (ipv4);
71   Ptr<Ipv4Private> ipv4Private = Create<Ipv4Private> (ipv4);
72
73   Object::AddInterface (ipv4Private);
74   Object::AddInterface (ipv4Impl);
75   Object::AddInterface (arpPrivate);
76   Object::AddInterface (udpImpl);
77   Object::AddInterface (l3Demux);
78   Object::AddInterface (ipv4L4Demux);
79 }
```

There are a few things to note in this function. First, lines 57-59 create instances of the layer-3 and layer-4 protocols, and assign their pointers to *ns-3*smart pointers. Note the use of the Create method, which is a templated wrapper around operator new. Each of these objects has a back-pointer ("this") to the InternetNode. (**Note:** This class does not make use yet of the replaceable component system; objects are created with raw Create functions.)

Next, lines 61-62 create demultiplexers (demuxes). These are analogous to *ns-2*Classifiers, and they direct packets to the right layer-3 or layer-4 protocol. These demuxes have to be told about what layer-3 protocols are active in a node (line 64-66). If/when we have Tcp and Ipv6 models, those will be added as well.

Lines 68-71 create some private implementation-related objects that make up the Ipv4, Arp, and Udp implementations.

Finally, we want to make interfaces to these objects available to the QueryInterface facility, so we explicitly add these pointers in lines 73-78.

The next two sections graphically depict how the various objects in the src/internet-node directory relate to one another.

5.2 Send packet processing chain

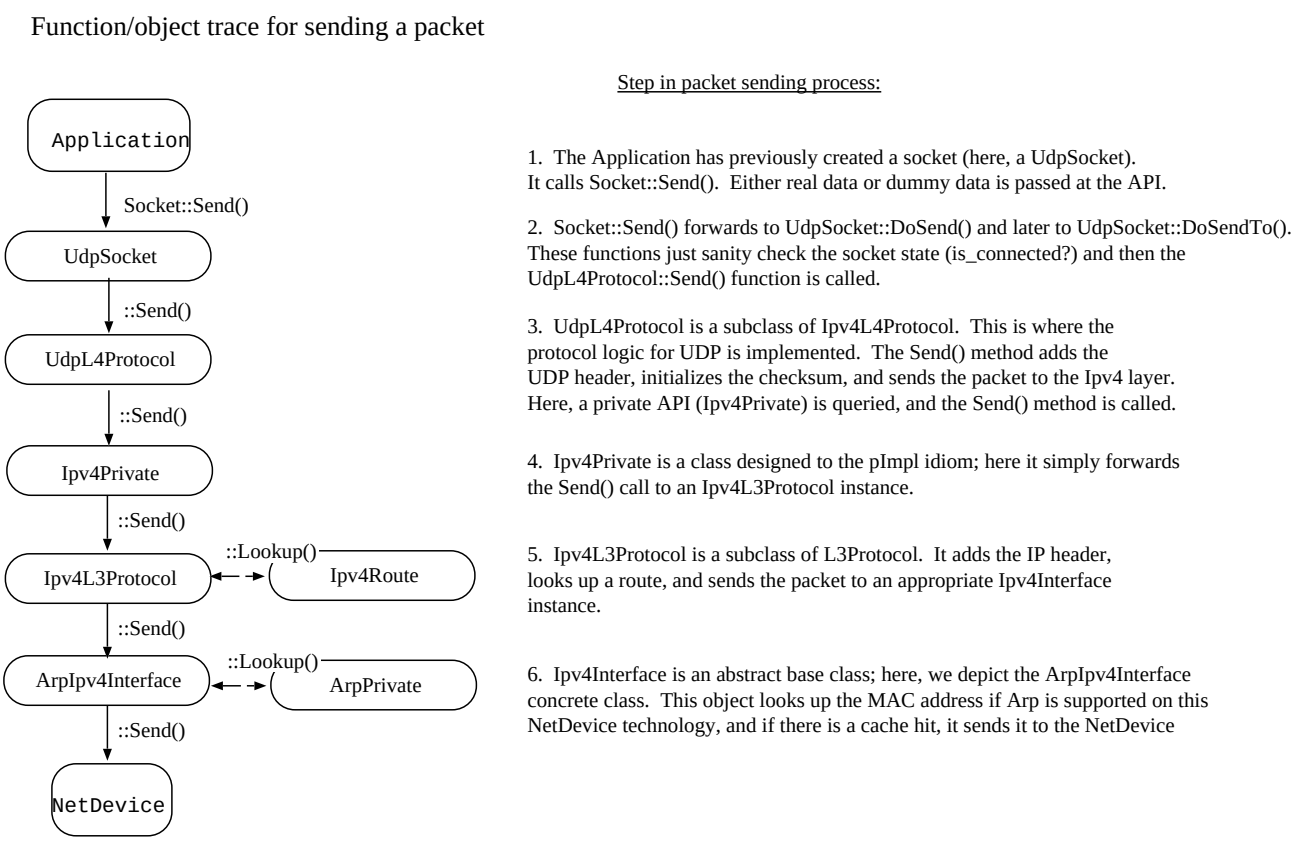

Figure 5.1: Steps in the send packet processing chain (Ipv4/UDP example).

5.3 Receive packet processing chain

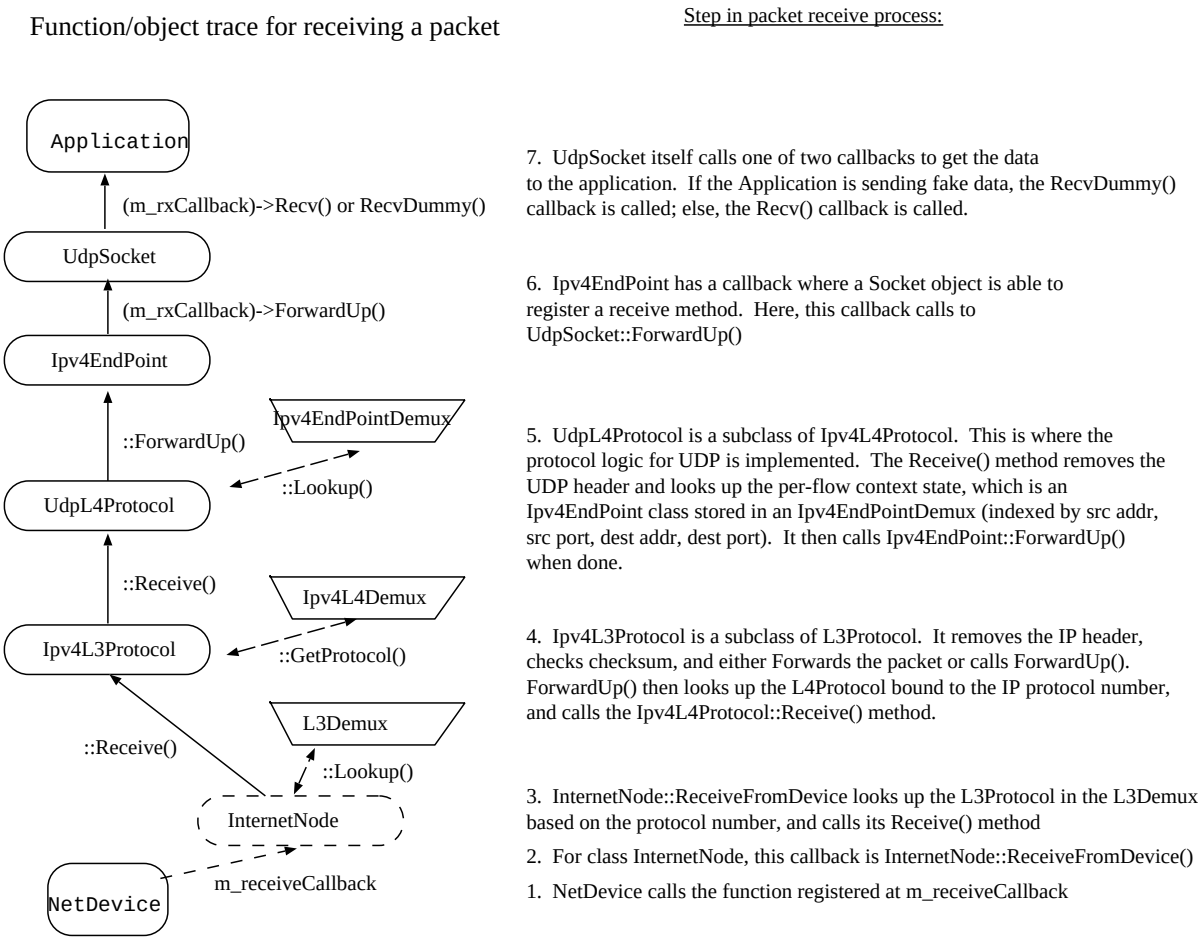

Figure 5.2: Steps in the receive packet processing chain (Ipv4/UDP example).

6 ns-3 routing

This writeup describes some goals and requirements for initial static routing, and a proposed routing roadmap.

Note: the ns-3.0.3 release does not include support for anything but manual static routing. Section 6.4 below shows the current routing roadmap.

6.1 Overview

Routing (including static routing) is a bit more complex than what is found in ns-2, which relies on a simple shortest path computation based on node ID or generic hierarchical addresses. Here, we want to account for the fact that our network may be composed of variably subnetted IP links, and we should offer some alternatives for defining link costs.

There are several components that need to be defined:

- unicast and multicast
- IPv4 and IPv6 (and others)
- routing manager object that can populate static routing tables
- routing API of a node
- forwarding implementation of a node

For starters, we focus on IPv4 and unicast, followed by IPv4 multicast.

For static routing, we propose a RouteManager object, which is a singleton object responsible for populating per-node forwarding tables. When a user calls "PopulateRoutingTables ()", the RouteManager will walk the topology and build a network map representation of the network, will run a shortest path computation, and will then access the routing API of each node and populate the forwarding tables. Users may enter per-node static routes, in the usual way, that override the values placed in the table by the RouteManager. The RouteManager must deal with point-to-point and shared links, must correctly deal with IP subnetting (and perhaps use different approaches for different address families), and must allow for different cost metric assignments, such as "every link is a single hop" or "link metrics are a function of the inverse of the bandwidth of the link".

For dynamic routing, we propose to port XORP or quagga rather than rewrite routing protocols for ns-3. There is some experience with this in previous simulators; most notably, GTNetS has supported ports of quagga BSD and OSPFv3.

The routing API of a node should be a queriable interface with some implementation independence. Ideally, we ought to be able to use the same interface for both Linux and idealized stacks, and for manual routing configs as well as dynamic routing protocols such as XORP that use routing sockets directly.

The node forwarding table is instantiated by some object in the node; either an IPv4 or IPv6-layer object, or something like a ported Linux stack. It should have the following properties:

- keep ARP and routing tables separate (otherwise, causes problems for parallel links between nodes)
- support multiple forwarding tables
- support for routing metrics (transport protocols), probably via clonable entries

Linux is probably the right implementation model for this, since BSD has mixed ARP and IP routing, unfortunately, and doesn't support multiple forwarding tables.

Users should be able to trace (either debug print, or redirect to a trace file) the routing table in a format such as used in an Unix implementation:

```
# netstat -nr (or # route -n)
Kernel IP routing table
Destination   Gateway      Genmask         Flags  MSS Window  irtt Iface
127.0.0.1     *            255.255.255.255 UH       0 0          0 lo
172.16.1.0    *            255.255.255.0   U        0 0          0 eth0
172.16.2.0    172.16.1.1   255.255.255.0   UG       0 0          0 eth0

# ip route show
192.168.99.0/24 dev eth0  scope link
127.0.0.0/8 dev lo  scope link
default via 192.168.99.254 dev eth0
```

Static multicast should be implemented as well. This would ignore group membership and ensure that a copy of every sourced multicast datagram would be delivered to each node. This might be implemented as an RPF mechanism that functioned on-demand by querying the forwarding table, and perhaps optimized by a small multicast forwarding cache. It is a bit trickier to implement over wireless links where the input interface is the same as the output interface; other aspects of the packet must be considered and the forwarding logic slightly changed to allow for forwarding out the same interface.

6.2 RouteManager

This is some type of static or singleton object that performs the following operations. It doesn't have to be particularly efficient or fast (at least initially) because it only executes once (per node) at the beginning of the simulation.

We should try to find applicable code for at least the SP computation sections below. There are plenty of Dijkstra examples on the web. GTNetS RoutingStatic class also probably could be used as a basis for this.

```
enum AddressFamily {
  Ipv4,
  Ipv6,
  ...
};

enum RouteMetric {
  UnitCost,
  InverseBandwidth,
  ...
};

// public function
bool
RouteManager::ComputeRoutes(Ptr<Node> n, uint8_t addressFamily, uint8_t routeMetric) {
  /*
   walkNodeListToBuildTopology(addressFamily, routeMetric);
```

```
   useDijkstraSPToCreateRoutes(n);
   populateNodeTables(n);
   */
}

// public (non-member?) function
bool
ComputeAllRoutes(uint8_t addressFamily, uint8_t routeMetric) {
   /*
   for each Node in the NodeList {
      RouteManager::ComputeRoutes(n, addressFamily, routeMetric);
   }
   */
}
```

6.3 Node routing interface

```
//
// Interface for static routing
// - interface class patterned after route(8) (Linux man page)
// - the two main operations are "add" and "del"
//
// route [-v] [-A family] add [-net|-host] target [netmask Nm] [gw Gw]
// [metric N] [mss M] [window W] [irtt I] [reject] [mod] [dyn]
// [reinstate] [[dev] If]
//
// route [-v] [-A family] del [-net|-host] target [gw Gw] [netmask Nm]
// [metric N] [[dev] If]
//

class Ipv4RouteMetrics
{
  // This will be a class to allow to pass route metrics across the API,
  // such as "metric", "mss", "window", "irtt", "flags (mod, dyn, reinstate)
}

class Ipv4RouteObject;  // analogous to routing struct in kernel that holds
                        // state for the route; used to fetch at the API

class Ipv4Route : public Object
{
public:
  static const InterfaceId iid;

  virtual void AddHostRoute (Ipv4Address target,
                             Ipv4Mask netmask,
                             Ipv4Address gateway) = 0;

  virtual void AddHostRoute (Ipv4Address target,
                             Ipv4Mask netmask,
                             uint32_t interface) = 0;
```

```
virtual void AddNetworkRoute (Ipv4Address target,
                              Ipv4Mask netmask,
                              Ipv4Address gateway) = 0;

virtual void AddNetworkRoute (Ipv4Address target,
                              Ipv4Mask netmask,
                              uint32_t interface) = 0;

// Define future variants of the above that allow an Ipv4RouteMetrics to
// also be passed

virtual void AddDefaultRoute (Ipv4Address gateway) = 0;

virtual void AddDefaultRoute (Ipv4Address gateway,
                              uint32_t interface) = 0;


virtual void DelHostRoute (Ipv4Address target) = 0;

virtual void DelHostRoute (Ipv4Address target,
                           Ipv4Mask netmask,
                           Ipv4Address gateway) = 0;

virtual void DelHostRoute (Ipv4Address target,
                           Ipv4Mask netmask,
                           uint32_t interface) = 0;

virtual void DelNetworkRoute (Ipv4Address target,
                              Ipv4Mask netmask) = 0;

virtual void DelNetworkRoute (Ipv4Address target,
                              Ipv4Mask netmask,
                              Ipv4Address gateway) = 0;

virtual void DelNetworkRoute (Ipv4Address target,
                              Ipv4Mask netmask,
                              uint32_t interface) = 0;

virtual void DelDefaultRoute (void) = 0;

virtual void DelInterfaceRoutes (uint32_t interface) = 0;

/**
 * \returns the number of entries in the routing table.
 */
virtual uint32_t GetNRoutes (void) = 0;
/**
 * \param i index of route to return
 * \returns the route whose index is i
 */
virtual Ipv4RouteObject *GetRoute (uint32_t i) = 0;
```

6.4 Proposed roadmap

- **June 15:** This writeup
- **July 15:** Support IPv4 static routing with PointToPoint numbered links
- **August 15:** Extend IPv4 static routing to Ethernet (shared links), add static multicast forwarding over Ethernet and PointToPoint
- **Sept 15:** Add static multicast forwarding over wireless interface

In parallel, work on bringing XORP or quagga routing to *ns-3*, but it will take several months to port and enable.

There are presently no roadmap plans for IPv6.