ns-3 Reference Manual

ns-3 project
feedback: ns-developers@isi.edu

19 December 2008

This is an `ns-3` reference manual. Primary documentation for the `ns-3` project is available in four forms:

- ns-3 Tutorial
- ns-3 Doxygen: Documentation of the public APIs of the simulator
- Reference Manual (this document)
- ns-3 wiki

This document is written in GNU Texinfo 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.

This software is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see `http://www.gnu.org/licenses/`.

Table of Contents

1 Random variables

ns-3 contains a built-in pseudo-random number generator (PRNG). It is important for serious users of the simulator to understand the functionality, configuration, and usage of this PRNG, and to decide whether it is sufficient for his or her research use.

1.1 Quick Overview

ns-3 random numbers are provided via instances of `class RandomVariable`.

- **by default, ns-3 simulations use a random seed**; if there is any randomness in the simulation, each run of the program will yield different results. To use a fixed seed, users must call `RandomVariable::UseGlobalSeed ()` at the beginning of the program; see section See Section 1.3 [Seeding and independent replications], page 2
- each RandomVariable used in ns-3 has a virtual random number generator associated with it; all random variables use either a fixed or random seed based on the use of the global seed (previous bullet);
- if you intend to perform multiple runs of the same scenario, with different random numbers, please be sure to read the section on how to perform independent replications: See Section 1.3 [Seeding and independent replications], page 2.

Read further for more explanation about the random number facility for ns-3.

1.2 Background

Simulations use a lot of random numbers; the study in [cite] found that most network simulations spend as much as 50% of the CPU generating random numbers. Simulation users need to be concerned with the quality of the (pseudo) random numbers and the independence between different streams of random numbers.

Users need to be concerned with a few issues, such as:

- the seeding of the random number generator and whether a simulation run is deterministic or not,
- how to acquire different streams of random numbers that are independent from one another, and
- how long it takes for streams to cycle

We will introduce a few terms here: a RNG provides a long sequence of (pseudo) random numbers. The length of this sequence is called the *cycle length* or *period*, after which the RNG will repeat itself. This sequence can be partitioned into disjoint *streams*. A stream of a RNG is a contiguous subset or block of the RNG sequence. For instance, if the RNG period is of length N, and two streams are provided from this RNG, then the first stream might use the first N/2 values and the second stream might produce the second N/2 values. An important property here is that the two streams are uncorrelated. Likewise, each stream can be partitioned disjointly to a number of uncorrelated *substreams*. The underlying RNG hopefully produces a pseudo-random sequence of numbers with a very long cycle length, and partitions this into streams and substreams in an efficient manner.

ns-3 uses the same underlying random number generator as does ns-2: the MRG32k3a generator from Pierre L'Ecuyer. A detailed description can be found

in http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf. The MRG32k3a generator provides 1.8x10^19 independent streams of random numbers, each of which consists of 2.3x10^15 substreams. Each substream has a period (*i.e.*, the number of random numbers before overlap) of 7.6x10^22. The period of the entire generator is 3.1x10^57. Figure ref-streams provides a graphical idea of how the streams and substreams fit together.

Class `ns3::RandomVariable` is the public interface to this underlying random number generator. When users create new RandomVariables (such as UniformVariable, ExponentialVariable, etc.), they create an object that uses one of the distinct, independent streams of the random number generator. Therefore, each object of type RandomVariable has, conceptually, its own "virtual" RNG. Furthermore, each RandomVariable can be configured to use one of the set of substreams drawn from the main stream.

An alternate implementation would be to allow each RandomVariable to have its own (differently seeded) RNG. However, we cannot guarantee as strongly that the different sequences would be uncorrelated in such a case; hence, we prefer to use a single RNG and streams and substreams from it.

1.3 Seeding and independent replications

ns-3 simulations can be configured to produce deterministic or random results. If the ns-3 simulation is configured to use a fixed, deterministic seed with the same run number, it should give the same output each time it is run.

By default, ns-3 simulations use random seeds where the seeding is drawn from `/dev/random` (if it is available) or else from the time of day. A user who wants to fix the initial seeding of the PRNG must call the following static method during simulation configuration:

```
RandomVariable::UseGlobalSeed (uint32_t s0, s1, s2, s3, s4, s5);
```

where the six parameters are each of type uint32_t.

A typical use case is to run a simulation as a sequence of independent trials, so as to compute statistics on a large number of independent runs. The user can either change the global seed and rerun the simulation, or can advance the substream state of the RNG. This seeding and substream state setting must be called before any random variables are created; e.g.

```
RandomVariable::UseGlobalSeed(1,2,3,4,5,6);
int N = atol(argv[1]); //read in run number from command line
RandomVariable::SetRunNumber(N);
// Now, create random variables
UniformVariable x(0,10);
ExponentialVariable y(2902);
...
```

Which is better, setting a new seed or advancing the substream state? There is no guarantee that the streams produced by two random seeds will not overlap. The only way to guarantee that two streams do not overlap is to use the substream capability provided by the RNG implementation. **Therefore, use the substream capability to produce multiple independent runs of the same simulation.** In other words, the more statistically rigorous way

to configure multiple independent replications is not to simply ignore the seeding (and use /dev/random to seed the generator each time) but instead to use a fixed seed and to iterate the run number. This implementation allows for a maximum of 2.3x10^15 independent replications using the substreams.

1.4 class RandomVariable

All random variables should derive from `class RandomVariable`. This base class provides a few static methods for globally configuring the behavior of the random number generator. Derived classes provide API for drawing random variates from the particular distribution being supported.

Each RandomVariable created in the simulation is given a generator that is a new RNGStream from the underlying PRNG. Used in this manner, the L'Ecuyer implementation allows for a maximum of 1.8x10^19 random variables. Each random variable in a single replication can produce up to 7.6x10^22 random numbers before overlapping.

1.5 Base class public API

Below are excerpted a few public methods of `class RandomVariable` that deal with the global configuration and state of the RNG.

```
/**
 * \brief Set seeding behavior
 *
 * Specify whether the POSIX device /dev/random is to
 * be used for seeding.  When this is used, the underlying
 * generator is seeded with data from /dev/random instead of
 * being seeded based upon the time of day.   Defaults to true.
 */
static  void UseDevRandom(bool udr = true);

 /**
 * \brief Use the global seed to force precisely reproducible results.
 */
static void UseGlobalSeed(uint32_t s0, uint32_t s1, uint32_t s2,
                          uint32_t s3, uint32_t s4, uint32_t s5);

/**
 * \brief Set the run number of this simulation
 */
static void SetRunNumber(uint32_t n);

/**
 * \brief Get the internal state of the RNG
 *
 * This function is for power users who understand the inner workings
 * of the underlying RngStream method used.  It returns the internal
 * state of the RNG via the input parameter.
```

```
   * \param seed Output parameter; gets overwritten with the internal state
   * of the RNG.
   */
  void GetSeed(uint32_t seed[6]) const;
```

We have already described the seeding configuration above.

1.6 Types of RandomVariables

The following types of random variables are provided, and are documented in the ns-3 Doxygen or by reading `src/core/random-variable.h`. Users can also create their own custom random variables by deriving from class RandomVariable.

- `class UniformVariable`
- `class ConstantVariable`
- `class SequentialVariable`
- `class ExponentialVariable`
- `class ParetoVariable`
- `class WeibullVariable`
- `class NormalVariable`
- `class EmpiricalVariable`
- `class IntEmpiricalVariable`
- `class DeterministicVariable`
- `class LogNormalVariable`
- `class TriangularVariable`

1.7 Semantics of RandomVariable objects

RandomVariable objects have value semantics. This means that they can be passed by value to functions. The can also be passed by reference to const. RandomVariables do not derive from `ns3::Object` and we do not use smart pointers to manage them; they are either allocated on the stack or else users explicitly manage any heap-allocated RandomVariables.

RandomVariable objects can also be used in ns-3 attributes, which means that values can be set for them through the ns-3 attribute system. An example is in the propagation models for WifiNetDevice:

```
TypeId
RandomPropagationDelayModel::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::RandomPropagationDelayModel")
    .SetParent<PropagationDelayModel> ()
    .AddConstructor<RandomPropagationDelayModel> ()
    .AddAttribute ("Variable",
                   "The random variable which generates random delays (s).",
                   RandomVariableValue (UniformVariable (0.0, 1.0)),
        MakeRandomVariableAccessor (&RandomPropagationDelayModel::m_variable),
                   MakeRandomVariableChecker ())
```

```
    ;
  return tid;
}
```

Here, the ns-3 user can change the default random variable for this delay model (which is a UniformVariable ranging from 0 to 1) through the attribute system.

1.8 Using other PRNG

There is presently no support for substituting a different underlying random number generator (e.g., the GNU Scientific Library or the Akaroa package). Patches are welcome.

1.9 More advanced usage

To be completed

1.10 Publishing your results

When you publish simulation results, a key piece of configuration information that you should always state is how you used the the random number generator.

- what seeds you used,
- what RNG you used if not the default,
- how were independent runs performed,
- for large simulations, how did you check that you did not cycle.

It is incumbent on the researcher publishing results to include enough information to allow others to reproduce his or her results. It is also incumbent on the researcher to convince oneself that the random numbers used were statistically valid, and to state in the paper why such confidence is assumed.

1.11 Summary

Let's review what things you should do when creating a simulation.

- Decide whether you are running with a fixed seed or random seed; a random seed is the default,
- Decide how you are going to manage independent replications, if applicable,
- Convince yourself that you are not drawing more random values than the cycle length, if you are running a long simulation, and
- When you publish, follow the guidelines above about documenting your use of the random number generator.

The program *samples/main-random.cc* has some examples of usage.

2 Callbacks

Some new users to ns-3 are unfamiliar with an extensively used programming idiom used throughout the code: the "ns-3 callback". This chapter provides some motivation on the callback, guidance on how to use it, and details on its implementation.

2.1 Motivation

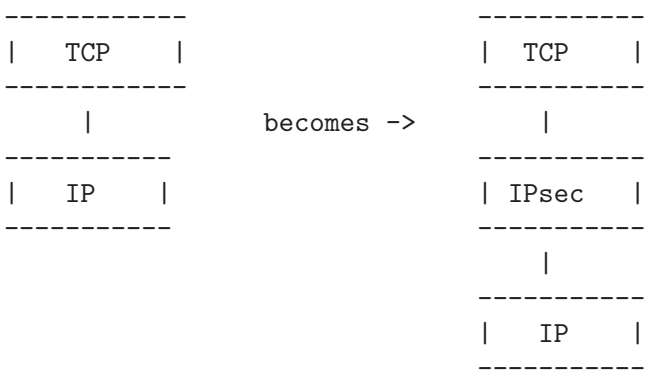
Consider that you have two simulation models A and B, and you wish to have them pass information between them during the simulation. One way that you can do that is that you can make A and B each explicitly knowledgable about the other, so that they can invoke methods on each other.

```
class A {
public:
  void ReceiveInput ( // parameters );
  ...
}

(in another source file:)

class B {
public:
  void ReceiveInput ( // parameters);
  void DoSomething (void);
  ...

private:
  A* a_instance; // pointer to an A
}

void
B::DoSomething()
{
  // Tell a_instance that something happened
  a_instance->ReceiveInput ( // parameters);
  ...
}
```

This certainly works, but it has the drawback that it introduces a dependency on A and B to know about the other at compile time (this makes it harder to have independent compilation units in the simulator) and is not generalized; if in a later usage scenario, B needs to talk to a completely different C object, the source code for B needs to be changed to add a "c_instance" and so forth. It is easy to see that this is a brute force mechanism of communication that can lead to programming cruft in the models.

This is not to say that objects should not know about one another if there is a hard dependency between them, but that often the model can be made more flexible if its interactions are less constrained at compile time.

This is not an abstract problem for network simulation research, but rather it has been a source of problems in previous simulators, when researchers want to extend or modify the system to do different things (as they are apt to do in research). Consider, for example, a user who wants to add an IPsec security protocol sublayer between TCP and IP:



If the simulator has made assumptions, and hard coded into the code, that IP always talks to a transport protocol above, the user may be forced to hack the system to get the desired interconnections.

An alternative that provides this flexibility is to use a level of indirection that is commonly known in programming as a callback. A callback function is not invoked explicitly by the caller but is rather delegated to another function that receives the callback function's address and can call it.

You may be familiar with function pointers in C or C++; these can be used to implement callbacks. For more information on introductory callbacks, an online reference is: Declaring Function Pointers and Implementing Callbacks and Callback (computer science)–Wikipedia.

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. It is therefore more type-safe to use than traditional function pointers, but the syntax may look imposing at first. This section is designed to walk you through the callback system so that you can be comfortable using it in `ns-3`.

2.2 Using the Callback API

The Callback API is fairly 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.

This is best observed via walking through an example, based on `samples/main-callback.cc`.

2.2.1 Using the Callback API with static functions

Consider a function:

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

Consider also the following main program snippett:

```
int main (int argc, char *argv[])
{
  // return type: double
  // first arg type: double
  // second arg type: double
  Callback<double, double, double> one;
}
```

This class template Callback implements what is known as the Functor Design Pattern. It is used to declare the type of a callback. It contains one mandatory argument (the return type of the function to be assigned to this callback) and up to five optional arguments, which each specify the type of the arguments (if your function has more than five arguments, then this can be handled by extending the callback implementation).

So in the above, we have a declared a callback named "one" that will eventually hold a function pointer. The function that it will hold must return double and must support two double arguments. If one tries to pass a function whose signature does not match the declared callback, the compilation will fail.

Now, we need to tie together this callback instance and the actual target function (CbOne). Notice above that CbOne has the same function signature types as the callback– this is important. We can pass in any such properly-typed function to this callback. Let's look at this more closely:

```
static double CbOne (double a, double b) {}
          ^           ^          ^
          |        ---|    ------|
          |        |       |
Callback<double, double, double> one;
```

You can only bind a function to a callback if they have the matching signature. The first template argument is the return type, and the additional template arguments are the types of the arguments of the function signature.

Now, let's bind our callback "one" to the function that matches its signature:

```
  // build callback instance which points to cbOne function
  one = MakeCallback (&CbOne);
```

Then, later in the program, if the callback is to be used, it can be used as follows:

```
// this is not a null callback
  NS_ASSERT (!one.IsNull ());
  // invoke cbOne function through callback instance
```

```
  double retOne;
  retOne = one (10.0, 20.0);
```

The check `IsNull()` ensures that the callback is not null; that there is a function to call behind this callback. Then, `one()` returns the same result as if `CbOne()` had been called directly.

2.2.2 Using the Callback API with member functions

Generally, you will not be calling static functions but instead public member functions of an object. In this case, an extra argument is needed to the MakeCallback function, to tell the system on which object the function should be invoked. Consider this example, also from main-callback.cc:

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

int main ()
{
  ...
  // 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);
  ...
}
```

Here, we pass a (raw) pointer to the `MakeCallback<>` function, that says, when `two ()` is invoked, to call the `CbTwo` function on the object pointed to by `&cb`.

A variation of this is used when objects are referred to by ns-3 smart pointers. The MakeCallback API takes a raw pointer, so we need to call `PeekPointer ()` to obtain this raw pointer. So the example above would look like:

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

int main ()
{
  ...
```

```
  // return type: int
  // first arg type: double
  Callback<int, double> two;
  Ptr<MyCb> cb = CreateObject<MyCb> ();
  // build callback instance which points to MyCb::cbTwo
  two = MakeCallback (&MyCb::CbTwo, PeekPointer (cb));
  ...
}
```

2.2.3 Building Null Callbacks

It is possible for callbacks to be null; hence it may be wise to check before using them. There is a special construct for a null callback, which is preferable to simply passing "0" as an argument; it is the `MakeNullCallback<>` construct:

```
  two = MakeNullCallback<int, double> ();
  // invoking a null callback is just like
  // invoking a null function pointer:
  // it will crash at runtime.
  //int retTwoNull = two (20.0);
  NS_ASSERT (two.IsNull ());
```

2.3 Callback locations in ns-3

Where are callbacks frequently used in ns-3? Here are some of the more visible ones to typical users:

2.3.1 Socket API

2.3.2 Layer-2/Layer-3 API

2.3.3 Tracing subsystem

2.3.4 Routing

Route Reply

2.4 Implementation details

This section is advanced explanation for C++ experts interested in the implementation, and may be skipped by most users.

This code was originally written based on the techniques described here. It was subsequently rewritten to follow the architecture outlined in Modern C++ Design: Generic Programming and Design Patterns Applied– Alexandrescu, chapter 5, "Generalized Functors".

This code uses:

- default template parameters to saves users from having to specify empty parameters when the number of parameters is smaller than the maximum supported number
- the pimpl idiom: the Callback class is passed around by value and delegates the crux of the work to its pimpl pointer.

- two pimpl implementations which derive from CallbackImpl FunctorCallbackImpl can be used with any functor-type while MemPtrCallbackImpl can be used with pointers to member functions.
- a reference list implementation to implement the Callback's value semantics.

This code most notably departs from the Alexandrescu implementation in that it does not use type lists to specify and pass around the types of the callback arguments. Of course, it also does not use copy-destruction semantics and relies on a reference list rather than autoPtr to hold the pointer.

3 Attributes

In ns-3 simulations, there are two main aspects to configuration:

- the simulation topology and how objects are connected
- the values used by the models instantiated in the topology

This chapter focuses on the second item above: how the many values in use in ns-3 are organized, documented, and modifiable by ns-3 users. The ns-3 attribute system is also the underpinning of how traces and statistics are gathered in the simulator.

Before delving into details of the attribute value system, it will help to review some basic properties of `class ns3::Object`.

3.1 Object Overview

ns-3 is fundamentally a C++ object-based system. By this we mean that new C++ classes (types) can be declared, defined, and subclassed as usual.

Many ns-3 objects inherit from the `ns3::Object` base class. These objects have some additional properties that we exploit for organizing the system and improving the memory management of our objects:

- a "metadata" system that links the class name to a lot of meta-information about the object, including the base class of the subclass, the set of accessible constructors in the subclass, and the set of "attributes" of the subclass
- a reference counting smart pointer implementation, for memory management.

ns-3 objects that use the attribute system derive from either `ns3::Object` or `ns3::ObjectBase`. Most ns-3 objects we will discuss derive from `ns3::Object`, but a few that are outside the smart pointer memory management framework derive from `ns3::ObjectBase`.

Let's review a couple of properties of these objects.

3.1.1 Smart pointers

As introduced in the ns-3 tutorial, ns-3 objects are memory managed by a reference counting smart pointer implementation, `class ns3::Ptr`.

Smart pointers are used extensively in the ns-3 APIs, to avoid passing references to heap-allocated objects that may cause memory leaks. For most basic usage (syntax), treat a smart pointer like a regular pointer:

```
Ptr<WifiNetDevice> nd = ...;
nd->CallSomeFunction ();
// etc.
```

3.1.2 CreateObject

As we discussed above in ⟨undefined⟩ [Object Creation], page ⟨undefined⟩, at the lowest-level API, objects of type `ns3::Object` are not instantiated using `operator new` as usual but instead by a templated function called `CreateObject()`.

A typical way to create such an object is as follows:

```
Ptr<WifiNetDevice> nd = CreateObject<WifiNetDevice> ();
```

You can think of this as being functionally equivalent to:

```
WifiNetDevice* nd = new WifiNetDevice ();
```

Objects that derive from `ns3::Object` must be allocated on the heap using CreateObject(). Those deriving from `ns3::ObjectBase`, such as ns-3 helper functions and packet headers and trailers, can be allocated on the stack.

In some scripts, you may not see a lot of CreateObject() calls in the code; this is because there are some helper objects in effect that are doing the CreateObject()s for you.

3.1.3 TypeId

ns-3 classes that derive from class ns3::Object can include a metadata class called `TypeId` that records meta-information about the class, for use in the object aggregation and component manager systems:

- a unique string identifying the class
- the base class of the subclass, within the metadata system
- the set of accessible constructors in the subclass

3.1.4 Object Summary

Putting all of these concepts together, let's look at a specific example: `class ns3::Node`.

The public header file node.h has a declaration that includes a static GetTypeId function call:

```
class Node : public Object
{
public:
  static TypeId GetTypeId (void);
  ...
```

This is defined in the node.cc file as follows:

```
TypeId
Node::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::Node")
    .SetParent<Object> ()
    ;
  return tid;
}
```

Finally, when users want to create Nodes, they call:

```
Ptr<Node> n = CreateObject<Node> ();
```

We next discuss how attributes (values associated with member variables or functions of the class) are plumbed into the above TypeId.

3.2 Attribute Overview

The goal of the attribute system is to organize the access of internal member objects of a simulation. This goal arises because, typically in simulation, users will cut and paste/modify existing simulation scripts, or will use higher-level simulation constructs, but often will be interested in studying or tracing particular internal variables. For instance, use cases such as:

- "I want to trace the packets on the wireless interface only on the first access point"
- "I want to trace the value of the TCP congestion window (every time it changes) on a particular TCP socket"
- "I want a dump of all values that were used in my simulation."

Similarly, users may want fine-grained access to internal variables in the simulation, or may want to broadly change the initial value used for a particular parameter in all subsequently created objects. Finally, users may wish to know what variables are settable and retrievable in a simulation configuration. This is not just for direct simulation interaction on the command line; consider also a (future) graphical user interface that would like to be able to provide a feature whereby a user might right-click on an node on the canvas and see a hierarchical, organized list of parameters that are settable on the node and its constituent member objects, and help text and default values for each parameter.

3.2.1 Functional overview

We provide a way for users to access values deep in the system, without having to plumb accessors (pointers) through the system and walk pointer chains to get to them. Consider a class DropTailQueue that has a member variable that is an unsigned integer `m_maxPackets`; this member variable controls the depth of the queue.

If we look at the declaration of DropTailQueue, we see the following:

```
class DropTailQueue : public Queue {
public:
  static TypeId GetTypeId (void);
  ...

private:
  std::queue<Ptr<Packet> > m_packets;
  uint32_t m_maxPackets;
};
```

Let's consider things that a user may want to do with the value of m_maxPackets:

- Set a default value for the system, such that whenever a new DropTailQueue is created, this member is initialized to that default.
- Set or get the value on an already instantiated queue.

The above things typically require providing Set() and Get() functions, and some type of global default value.

In the ns-3 attribute system, these value definitions and accessor functions are moved into the TypeId class; e.g.:

```
TypeId DropTailQueue::GetTypeId (void)
```

```
{
  static TypeId tid = TypeId ("ns3::DropTailQueue")
    .SetParent<Queue> ()
    .AddConstructor<DropTailQueue> ()
    .AddAttribute ("MaxPackets",
                   "The maximum number of packets accepted by this DropTailQueue.",
                   UintegerValue (100),
                   MakeUintegerAccessor (&DropTailQueue::m_maxPackets),
                   MakeUintegerChecker<uint32_t> ())
    ;

  return tid;
}
```

The AddAttribute() method is performing a number of things with this value:

- Binding the variable m_maxPackets to a string "MaxPackets"
- Providing a default value (100 packets)
- Providing some help text defining the value
- Providing a "checker" (not used in this example) that can be used to set bounds on the allowable range of values

The key point is that now the value of this variable and its default value are accessible in the attribute namespace, which is based on strings such as "MaxPackets" and TypeId strings. In the next section, we will provide an example script that shows how users may manipulate these values.

3.2.2 Basic usage

Let's look at how a user script might access these values. This is based on the script found at `samples/main-attribute-value.cc`, with some details stripped out.

```
//
// This is a basic example of how to use the attribute system to
// set and get a value in the underlying system; namely, an unsigned
// integer of the maximum number of packets in a queue
//

int
main (int argc, char *argv[])
{

  // By default, the MaxPackets attribute has a value of 100 packets
  // (this default can be observed in the function DropTailQueue::GetTypeId)
  //
  // Here, we set it to 80 packets.  We could use one of two value types:
  // a string-based value or a Uinteger value
  Config::SetDefault ("ns3::DropTailQueue::MaxPackets", StringValue ("80"));
  // The below function call is redundant
  Config::SetDefault ("ns3::DropTailQueue::MaxPackets", UintegerValue (80));
```

```
// Allow the user to override any of the defaults and the above
// SetDefaults() at run-time, via command-line arguments
CommandLine cmd;
cmd.Parse (argc, argv);
```

The main thing to notice in the above are the two calls to `Config::SetDefault`. This is how we set the default value for all subsequently instantiated DropTailQueues. We illustrate that two types of Value classes, a StringValue and a UintegerValue class, can be used to assign the value to the attribute named by "ns3::DropTailQueue::MaxPackets".

Now, we will create a few objects using the low-level API; here, our newly created queues will not have a m_maxPackets initialized to 100 packets but to 80 packets, because of what we did above with default values.

```
Ptr<Node> n0 = CreateObject<Node> ();

Ptr<PointToPointNetDevice> net0 = CreateObject<PointToPointNetDevice> ();
n0->AddDevice (net0);

Ptr<Queue> q = CreateObject<DropTailQueue> ();
net0->AddQueue(q);
```

At this point, we have created a single node (Node 0) and a single PointToPointNetDevice (NetDevice 0) and added a DropTailQueue to it.

Now, we can manipulate the MaxPackets value of the already instantiated DropTailQueue. Here are various ways to do that.

3.2.2.1 Pointer-based access

We assume that a smart pointer (Ptr) to a relevant network device is in hand; here, it is the net0 pointer.

One way to change the value is to access a pointer to the underlying queue and modify its attribute.

First, we observe that we can get a pointer to the (base class) queue via the PointToPointNetDevice attributes, where it is called TxQueue

```
PointerValue tmp;
net0->GetAttribute ("TxQueue", tmp);
Ptr<Object> txQueue = tmp.GetObject ();
```

Using the GetObject function, we can perform a safe downcast to a DropTailQueue, where MaxPackets is a member

```
Ptr<DropTailQueue> dtq = txQueue->GetObject <DropTailQueue> ();
NS_ASSERT (dtq != 0);
```

Next, we can get the value of an attribute on this queue. We have introduced wrapper "Value" classes for the underlying data types, similar to Java wrappers around these types, since the attribute system stores values and not disparate types. Here, the attribute value is assigned to a UintegerValue, and the Get() method on this value produces the (unwrapped) uint32_t.

```
UintegerValue limit;
dtq->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("1.  dtq limit: " << limit.Get () << " packets");
```

Note that the above downcast is not really needed; we could have done the same using the Ptr<Queue> even though the attribute is a member of the subclass

```
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("2.  txQueue limit: " << limit.Get () << " packets");
```

Now, let's set it to another value (60 packets)

```
txQueue->SetAttribute("MaxPackets", UintegerValue (60));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("3.  txQueue limit changed: " << limit.Get () << " packets");
```

3.2.2.2 Namespace-based access

An alternative way to get at the attribute is to use the configuration namespace. Here, this attribute resides on a known path in this namespace; this approach is useful if one doesn't have access to the underlying pointers and would like to configure a specific attribute with a single statement.

```
Config::Set ("/NodeList/0/DeviceList/0/TxQueue/MaxPackets", UintegerValue (25));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("4.  txQueue limit changed through namespace: " <<
  limit.Get () << " packets");
```

We could have also used wildcards to set this value for all nodes and all net devices (which in this simple example has the same effect as the previous Set())

```
Config::Set ("/NodeList/*/DeviceList/*/TxQueue/MaxPackets", UintegerValue (15));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("5.  txQueue limit changed through wildcarded namespace: " <<
  limit.Get () << " packets");
```

3.2.3 Setting through constructors helper classes

Arbitrary combinations of attributes can be set and fetched from the helper and low-level APIs; either from the constructors themselves:

```
Ptr<Object> p = CreateObject<MyNewObject> ("n1", v1, "n2", v2, ...);
```

or from the higher-level helper APIs, such as:

```
mobility.SetPositionAllocator ("GridPositionAllocator",
                               "MinX", DoubleValue (-100.0),
                               "MinY", DoubleValue (-100.0),
                               "DeltaX", DoubleValue (5.0),
                               "DeltaY", DoubleValue (20.0),
                               "GridWidth", UintegerValue (20),
                               "LayoutType", StringValue ("RowFirst"));
```

3.2.4 Value classes

Readers will note the new FooValue classes which are subclasses of the AttributeValue base class. These can be thought of as an intermediate class that can be used to convert from

raw types to the Values that are used by the attribute system. Recall that this database is holding objects of many types with a single generic type. Conversions to this type can either be done using an intermediate class (IntegerValue, DoubleValue for "floating point") or via strings. Direct implicit conversion of types to Value is not really practical. So in the above, users have a choice of using strings or values:

```
p->Set ("cwnd", StringValue ("100")); // string-based setter
p->Set ("cwnd", IntegerValue (100)); // integer-based setter
```

The system provides some macros that help users declare and define new AttributeValue subclasses for new types that they want to introduce into the attribute system:

- ATTRIBUTE_HELPER_HEADER
- ATTRIBUTE_HELPER_CPP

3.3 Extending attributes

The ns-3 system will place a number of internal values under the attribute system, but undoubtedly users will want to extend this to pick up ones we have missed, or to add their own classes to this.

3.3.1 Adding an existing internal variable to the metadata system

Consider this variable in class TcpSocket:

```
uint32_t m_cWnd;   // Congestion window
```

Suppose that someone working with Tcp wanted to get or set the value of that variable using the metadata system. If it were not already provided by ns-3, the user could declare the following addition in the metadata system (to the TypeId declaration for TcpSocket):

```
    .AddParameter ("Congestion window",
                   "Tcp congestion window (bytes)",
                   Uinteger (1),
                   MakeUintegerAccessor (&TcpSocket::m_cWnd),
                   MakeUintegerChecker<uint16_t> ());
```

Now, the user with a pointer to the TcpSocket can perform operations such as setting and getting the value, without having to add these functions explicitly. Furthermore, access controls can be applied, such as allowing the parameter to be read and not written, or bounds checking on the permissible values can be applied.

3.3.2 Adding a new TypeId

Here, we discuss the impact on a user who wants to add a new class to ns-3; what additional things must be done to hook it into this system.

We've already introduced what a TypeId definition looks like:

```
TypeId
RandomWalk2dMobilityModel::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::RandomWalk2dMobilityModel")
    .SetParent<MobilityModel> ()
    .SetGroupName ("Mobility")
```

```
    .AddConstructor<RandomWalk2dMobilityModel> ()
    .AddAttribute ("Bounds",
                   "Bounds of the area to cruise.",
                   RectangleValue (Rectangle (0.0, 0.0, 100.0, 100.0)),
                   MakeRectangleAccessor (&RandomWalk2dMobilityModel::m_bounds),
                   MakeRectangleChecker ())
    .AddAttribute ("Time",
                   "Change current direction and speed after moving for this delay.",
                   TimeValue (Seconds (1.0)),
                   MakeTimeAccessor (&RandomWalk2dMobilityModel::m_modeTime),
                   MakeTimeChecker ())
    // etc (more parameters).
    ;
  return tid;
}
```

The declaration for this in the class declaration is one-line public member method:

```
 public:
  static TypeId GetTypeId (void);
```

Typical mistakes here involve:

- Not calling the SetParent method or calling it with the wrong type
- Not calling the AddConstructor method of calling it with the wrong type
- Introducing a typographical error in the name of the TypeId in its constructor
- Not using the fully-qualified c++ typename of the enclosing c++ class as the name of the TypeId

None of these mistakes can be detected by the ns-3 codebase so, users are advised to check carefully multiple times that they got these right.

3.4 Adding new class type to the attribute system

From the perspective of the user who writes a new class in the system and wants to hook it in to the attribute system, there is mainly the matter of writing the conversions to/from strings and attribute values. Most of this can be copy/pasted with macro-ized code. For instance, consider class Rectangle in the `src/mobility/` directory:

One line is added to the class declaration:

```
/**
 * \brief a 2d rectangle
 */
class Rectangle
{
...

};
```

One macro call and two operators, are added below the class declaration:

```
std::ostream &operator << (std::ostream &os, const Rectangle &rectangle);
```

```
std::istream &operator >> (std::istream &is, Rectangle &rectangle);

ATTRIBUTE_HELPER_HEADER (Rectangle);
```

In the class definition, the code looks like this:

```
ATTRIBUTE_HELPER_CPP (Rectangle);

std::ostream &
operator << (std::ostream &os, const Rectangle &rectangle)
{
  os << rectangle.xMin << "|" << rectangle.xMax << "|" << rectangle.yMin << "|" << rectangl
  return os;
}
std::istream &
operator >> (std::istream &is, Rectangle &rectangle)
 {
  char c1, c2, c3;
  is >> rectangle.xMin >> c1 >> rectangle.xMax >> c2 >> rectangle.yMin >> c3 >> rectangle.y
  if (c1 != '|' ||
      c2 != '|' ||
      c3 != '|')
    {
      is.setstate (std::ios_base::failbit);
    }
  return is;
}
```

These stream operators simply convert from a string representation of the Rectangle ("xMin|xMax|yMin|yMax") to the underlying Rectangle, and the modeler must specify these operators and the string syntactical representation of an instance of the new class.

3.5 ConfigStore

Feedback requested: This is an experimental feature of ns-3. It is not in the main tree. If you like this feature and would like to provide feedback on it, please email us.

Values for ns-3 attributes can be stored in an ascii text file and loaded into a future simulation. This feature is known as the ns-3 ConfigStore. The ConfigStore code is in `src/contrib/`. It is not yet main-tree code, because we are seeking some user feedback.

We can explore this system by using an example. Copy the `csma-bridge.cc` file to the scratch directory:

```
cp examples/csma-bridge.cc scratch/
./waf
```

Let's edit it to add the ConfigStore feature. First, add an include statement, and then add these lines:

```
#include "contrib-module.h"
...
int main (...)
```

```
{
  // setup topology

  // Invoke just before entering Simulator::Run ()
  ConfigStore config;
  config.Configure ();

  Simulator::Run ();
}
```

There is an attribute that governs whether the Configure() call either stores a simulation configuration in a file and exits, or whether it loads a simulation configuration file annd proceeds. First, the `LoadFilename` attribute is checked, and if non-empty, the program loads the configuration from the filename provided. If LoadFilename is empty, and if the `StoreFilename` attribute is populated, the configuration will be written to the output filename specified.

While it is possible to generate a sample config file and lightly edit it to change a couple of values, there are cases where this process will not work because the same value on the same object can appear multiple times in the same automatically-generated configuration file under different configuration paths.

As such, the best way to use this class is to use it to generate an initial configuration file, extract from that configuration file only the strictly necessary elements, and move these minimal elements to a new configuration file which can then safely be edited and loaded in a subsequent simulation run.

So, let's do that as an example. We'lll run the program once to create a configure file, and look at it. If you are running bash shell, the below command should work (which illustrates how to set an attribute from the command line):

```
./build/debug/scratch/csma-bridge --ns3::ConfigStore::StoreFilename=test.config
```

or, if the above does not work (the above requires rpath support), try this:

```
./waf --command-template="%s --ns3::ConfigStore::StoreFilename=test.config" --run scratch/c
```

Running the program should yield a "test.config" output configuration file that looks like this:

```
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/Addre
ss 00:00:00:00:00:01
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/Frame
Size 1518
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/SendE
nable true
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/Recei
veEnable true
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/TxQue
ue/$ns3::DropTailQueue/MaxPackets 100
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNetDevice/Mtu 1
500
...
```

The above lists, for each object in the script topology, the value of each registered attribute. The syntax of this file is that the unique name of the attribute (in the attribute namespace) is specified on each line, followed by a value.

This file is intended to be a convenient record of the parameters that were used in a given simulation run, and can be stored with simulation output files. Additionally, this file can also be used to parameterize a simulation, instead of editing the script or passing in command line arguments. For instance, a person wanting to run the simulation can examine and tweak the values in a pre-existing configuration file, and pass the file to the program. In this case, the relevant commands are:

```
./build/debug/scratch/csma-bridge --ns3::ConfigStore::LoadFilename=test.config
```

or, if the above does not work (the above requires rpath support), try this:

```
./waf --command-template="%s --ns3::ConfigStore::LoadFilename=test.config" --run scratch/cs
```

3.5.1 GTK-based ConfigStore

There is a GTK-based front end for the ConfigStore. This allows users to use a GUI to access and change variables. Screenshots of this feature are available in the ns-3 Overview presentation.

To use this feature, one must install libgtk and libgtk-dev; an example Ubuntu installation command is:

```
sudo apt-get install libgtk2.0-0 libgtk2.0-dev
```

To check whether it is configured or not, check the output of the ./waf configure step:

```
---- Summary of optional NS-3 features:
Threading Primitives          : enabled
Real Time Simulator           : enabled
GtkConfigStore                : not enabled (library 'gtk+-2.0 >= 2.12' not found)
```

In the above example, it was not enabled, so it cannot be used until a suitable version is installed and ./waf configure; ./waf is rerun.

Usage is almost the same as the non-GTK-based version:

```
// Invoke just before entering Simulator::Run ()
GtkConfigStore config;
config.Configure ();
```

Now, when you run the script, a GUI should pop up, allowing you to open menus of attributes on different nodes/objects, and then launch the simulation execution when you are done.

3.5.2 Future work

There are a couple of possible improvements:

- save a unique version number with date and time at start of file
- save rng initial seed somewhere.
- make each RandomVariable serialize its own initial seed and re-read it later
- add the default values

4 Object model

ns-3 is fundamentally a C++ object system. Objects can be declared and instantiated as usual, per C++ rules. ns-3 also adds some features to traditional C++ objects, as described below, to provide greater functionality and features. This manual chapter is intended to introduce the reader to the ns-3 object model.

This section describes the C++ class design for ns-3 objects. In brief, several 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, an object aggregation facility, and reference counting for memory management. Those familiar with component models such as COM or Bonobo will recognize elements of the design in the ns-3 object aggregation model, although the ns-3 design is not strictly in accordance with either.

4.1 Object-oriented behavior

C++ objects, in general, provide common object-oriented capabilities (abstraction, encapsulation, inheritance, and polymorphism) that are part of classic object-oriented design. ns-3 objects make use of these properties; for instance:

```
class Address
{
public:
  Address ();
  Address (uint8_t type, const uint8_t *buffer, uint8_t len);
  Address (const Address & address);
  Address &operator = (const Address &address);
  ...
private:
  uint8_t m_type;
  uint8_t m_len;
  ...
};
```

4.2 Object base classes

There are three special base classes used in ns-3. Classes that inherit from these base classes can instantiate objects with special properties. These base classes are:

- `class Object`
- `class ObjectBase`
- `class RefCountBase`

It is not required that ns-3 objects inherit from these class, but those that do get special properties. Classes deriving from `class Object` get the following properties.

- the ns-3 type and attribute system (see Chapter 3 [Attributes], page 12)
- an object aggregation system
- a smart-pointer reference counting system (class Ptr)

Classes that derive from `class ObjectBase` get the first two properties above, but do not get smart pointers. Classes that derive from `class RefCountBase` get only the smart-pointer reference counting system.

In practice, `class Object` is the variant of the three above that the ns-3 developer will most commonly encounter.

4.3 Memory management and class Ptr

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.

4.3.1 Reference counting smart pointer (Ptr)

Calling `Ref()` and `Unref()` all the time would be cumbersome, so ns-3 provides a smart pointer `class Ptr` 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.

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 CreateObject template functions to create the object and store it in a smart pointer to avoid memory leaks. These functions are really small convenience functions and their goal is just is save you a small bit of typing.

4.3.2 CreateObject and Create

Objects in C++ may be statically, dynamically, or automatically created. This holds true for ns-3 also, but some objects in the system have some additional frameworks available. Specifically, reference counted objects are usually allocated using a templated Create or CreateObject method, as follows.

For objects deriving from `class Object`:

```
Ptr<WifiNetDevice> device = CreateObject<WifiNetDevice> ();
```

Please do not create such objects using `operator new`; create them using `CreateObject()` instead.

For objects deriving from `class RefCountBase`, or other objects that support usage of the smart pointer class (in particular, the ns-3 Packet class), a templated helper function is available and recommended to be used:

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

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

4.3.3 Aggregation

The ns-3 object aggregation system is motivated in strong part by a recognition that a common use case for ns-2 has been the use of inheritance and 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 and GNOME Bonobo although full binary-level compatibility of replaceable components is not supported and we have tried to simplify the syntax and impact on model developers.

4.3.3.1 Aggregation example

`class Node` is a good example of the use of aggregation in ns-3. Note that there are not derived classes of Nodes in ns-3 such as class InternetNode. Instead, components (protocols) are aggregated to a node. Let's look at how some Ipv4 protocols are added to a node.

```
static void
AddIpv4Stack(Ptr<Node> node)
{
  Ptr<Ipv4L3Protocol> ipv4 = CreateObject<Ipv4L3Protocol> ();
  ipv4->SetNode (node);
  node->AggregateObject (ipv4);
  Ptr<Ipv4Impl> ipv4Impl = CreateObject<Ipv4Impl> ();
  ipv4Impl->SetIpv4 (ipv4);
  node->AggregateObject (ipv4Impl);
}
```

Note that the Ipv4 protocols are created using `CreateObject()`. Then, they are aggregated to the node. In this manner, the Node base class does not need to be edited to allow

users with a base class Node pointer to access the Ipv4 interface; users may ask the node for a pointer to its Ipv4 interface at runtime. How the user asks the node is described in the next subsection.

Note that it is a programming error to aggregate more than one object of the same type to an ns3::Object. So, for instance, aggregation is not an option for storing all of the active sockets of a node.

4.3.3.2 GetObject example

GetObject is a type-safe way to achieve a safe downcasting and to allow interfaces to be found on an object.

Consider a node pointer `m_node` that points to a Node object that has an implementation of IPv4 previously aggregated to it. 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:

```
Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4> ();
```

If the node in fact does not have an Ipv4 object aggregated to it, then the method will return null. Therefore, it is good practice to check the return value from such a function call. If successful, the user can now use the Ptr to the Ipv4 object that was previously aggregated to the node.

Another example of how one might use aggregation is to add optional models to objects. For in For instance, an existing Node object may have an "Energy Model" object aggregated to it at run time (without modifying and recompiling the node class). An existing model (such as a wireless net device) can then later "GetObject" 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. However, other nodes need not know anything about energy models.

We hope that this mode of programming will require much less need for developers to modify the base classes.

4.4 Downcasting

A question that has arisen several times is, "If I have a base class pointer (Ptr) to an object and I want the derived class pointer, should I downcast (via C++ dynamic cast) to get the derived pointer, or should I use the object aggregation system to `GetObject<> ()` to find a Ptr to the interface to the subclass API?"

The answer to this is that in many situations, both techniques will work. ns-3 provides a templated function for making the syntax of Object dynamic casting much more user friendly:

```
template <typename T1, typename T2>
Ptr<T1>
DynamicCast (Ptr<T2> const&p)
{
  return Ptr<T1> (dynamic_cast<T1 *> (PeekPointer (p)));
}
```

DynamicCast works when the programmer has a base type pointer and is testing against a subclass pointer. GetObject works when looking for different objects aggregated, but also

works with subclasses, in the same way as DynamicCast. If unsure, the programmer should use GetObject, as it works in all cases. If the programmer knows the class hierarchy of the object under consideration, it is more direct to just use DynamicCast.

5 Real-Time Scheduler

ns-3 has been designed for integration into testbed and virtual machine environments. To integrate with real network stacks and emit/consume packets, a real-time scheduler is needed to try to lock the simulation clock with the hardware clock. We describe here a component of this: the RealTime scheduler.

The purpose of the realtime scheduler is to cause the progression of the simulation clock to occur synchronously with respect to some external time base. Without the presence of an external time base (wall clock), simulation time jumps instantly from one simulated time to the next.

5.1 Behavior

When using a non-realtime scheduler (the default in ns-3), the simulator advances the simulation time to the next scheduled event. During event execution, simulation time is frozen. With the realtime scheduler, the behavior is similar from the perspective of simulation models (i.e., simulation time is frozen during event execution), but between events, the simulator will attempt to keep the simulation clock aligned with the machine clock.

When an event is finished executing, and the scheduler moves to the next event, the scheduler compares the next event execution time with the machine clock. If the next event is scheduled for a future time, the simulator sleeps until that realtime is reached and then executes the next event.

It may happen that, due to the processing inherent in the execution of simulation events, that the simulator cannot keep up with realtime. In such a case, it is up to the user configuration what to do. There are two ns-3 attributes that govern the behavior. The first is `ns3::RealTimeSimulatorImpl::SynchronizationMode`. The two entries possible for this attribute are `BestEffort` (the default) or `HardLimit`. In "BestEffort" mode, the simulator will just try to catch up to realtime by executing events until it reaches a point where the next event is in the (realtime) future, or else the simulation ends. In BestEffort mode, then, it is possible for the simulation to consume more time than the wall clock time. The other option "HardLimit" will cause the simulation to abort if the tolerance threshold is exceeded. This attribute is `ns3::RealTimeSimulatorImpl::HardLimit` and the default is 0.1 seconds.

A different mode of operation is one in which simulated time is **not** frozen during an event execution. This mode of realtime simulation was implemented but removed from the ns-3 tree because of questions of whether it would be useful. If users are interested in a realtime simulator for which simulation time does not freeze during event execution (i.e., every call to `Simulator::Now()` returns the current wall clock time, not the time at which the event started executing), please contact the ns-developers mailing list.

5.2 Usage

The usage of the realtime simulator is straightforward, from a scripting perspective. Users just need to set the attribute `SimulatorImplementationType` to the Realtime simulator, such as follows:

```
GlobalValue::Bind ("SimulatorImplementationType",
  StringValue ("ns3::RealtimeSimulatorImpl"));
```

There is a script in `examples/realtime-udp-echo.cc` that has an example of how to configure the realtime behavior. Try:

```
./waf --run realtime-udp-echo
```

Whether the simulator will work in a best effort or hard limit policy fashion is governed by the attributes explained in the previous section.

5.3 Implementation

The implementation is contained in the following files:

- `src/simulator/realtime-simulator-impl.cc,h`
- `src/simulator/wall-clock-synchronizer.cc,h`

In order to create a realtime scheduler, to a first approximation you just want to cause simulation time jumps to consume real time. We propose doing this using a combination of sleep- and busy- waits. Sleep-waits cause the calling process (thread) to yield the processor for some amount of time. Even though this specified amount of time can be passed to nanosecond resolution, it is actually converted to an OS-specific granularity. In Linux, the granularity is called a Jiffy. Typically this resolution is insufficient for our needs (on the order of a ten milliseconds), so we round down and sleep for some smaller number of Jiffies. The process is then awakened after the specified number of Jiffies has passed. At this time, we have some residual time to wait. This time is generally smaller than the minimum sleep time, so we busy-wait for the remainder of the time. This means that the thread just sits in a for loop consuming cycles until the desired time arrives. After the combination of sleep- and busy-waits, the elapsed realtime (wall) clock should agree with the simulation time of the next event and the simulation proceeds.

6 Emulation

ns-3 has been designed for integration into testbed and virtual machine environments. We have addressed this need by providing two kinds of net devices. The first kind, which we call an `Emu NetDevice` allows ns-3 simulations to send data on a "real" network. The second kind, called a `Tap NetDevice` allows a "real" host to participate in an ns-3 simulation as if it were one of the simulated nodes. An ns-3 simulation may be constructed with any combination of simulated, `Emu`, or `Tap` devices.

One of the use-cases we want to support is that of a testbed. A concrete example of an environment of this kind is the ORBIT testbed. ORBIT is a laboratory emulator/field trial network arranged as a two dimensional grid of 400 802.11 radio nodes. We integrate with ORBIT by using their "imaging" process to load and run ns-3 simulations on the ORBIT array. We use our `Emu NetDevice`s to drive the hardware in the testbed and we can accumulate results either using the ns-3 tracing and logging functions, or the native ORBIT data gathering techniques. See http://www.orbit-lab.org/ for details on the ORBIT testbed.

A simulation of this kind is shown in the following figure:

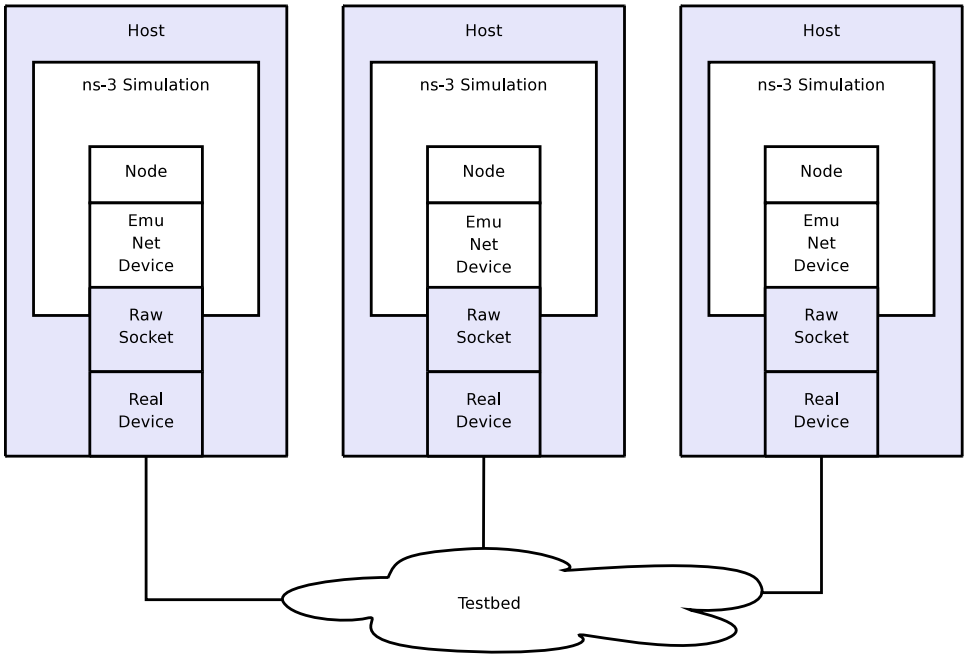


Figure 6.1

You can see that there are separate hosts, each running a subset of a "global" simulation. Instead of an ns-3 channel connecting the hosts, we use real hardware provided by the testbed. This allows ns-3 applications and protocol stacks attached to a simulation node to communicate over real hardware.

We expect the primary use for this configuration will be to generate repeatable experimental results in a real-world network environment that includes all of the ns-3 tracing, loging, visualization and statistics gathering tools.

In what can be viewed as essentially an inverse configuration, we allow “real” machines running native applications and protocol stacks to integrate with an ns-3 simulation. This allows for the simulation of large networks connected to a real mahince, and also enables virtualization. A simulation of this kind is shown in the following figure:

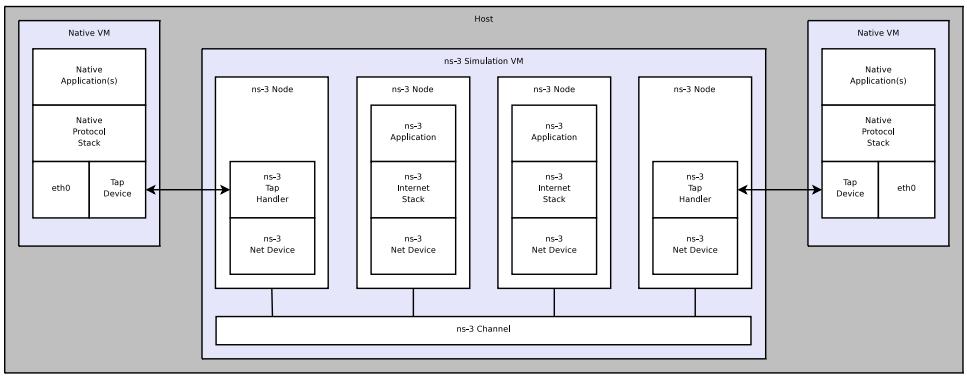


Figure 6.2: Implementation overview of emulated channel.

Here, you will see that there is a single host with a number of virtual machines running on it. An ns-3 simulation is shown running in the virtual machine shown in the center of the figure. This simulation has a number of nodes with associated ns-3 applications and protocol stacks that are talking to an ns-3 channel through native simulated ns-3 net devices.

There are also two virtual machines shown at the far left and far right of the figure. These VMs are running native (Linux) applications and protocol stacks. The VM is connected into the simulation by a Linux Tap net device. The user-mode handler for the Tap device is instantiated in the simulation and attached to a proxy node that represents the native VM in the simulation. These handlers allow the Tap devices on the native VMs to behave as if they were ns-3 net devices in the simulation VM. This, in turn, allows the native software and protocol suites in the native VMs to believe that they are connected to the simulated ns-3 channel.

We expect the typical use case for this environment will be to analyze the behavior of native applications and protocol suites in the presence of large simulated ns-3 networks.

6.1 Behavior

6.1.1 Emu Net Device

The `Emu` net device allows a simulation node to send and receive packets over a real network. The emulated net device relies on a specified interface being in promiscuous mode. It opens a raw socket and binds to that interface. We perform MAC spoofing to separate simulation network traffic from other network traffic that may be flowing to and from the host.

Normally, the use case for emulated net devices is in collections of small simulations that connect to the outside world through specific interfaces. For example, one could construct a number of virtual machines and connect them via a host-only network. To use the emulated

net device, you would need to set all of the host-only interfaces in promiscuous mode and provide an appropriate device name, "eth1" for example.

One could also use the Emu net device in a testbed situation where the host on which the simulation is running has a specific interface of interest which drives the testbed hardware. You would also need to set this specific interface into promiscuous mode and provide an appropriate device name to the ns-3 emulated net device. An example of this environment is the ORBIT testbed as described above.

The Emu net device only works if the underlying interface is up and in promiscuous mode. Packets will be sent out over the device, but we use MAC spoofing. The MAC addresses will be generated (by default) using the Organizationally Unique Identifier (OUI) 00:00:00 as a base. This vendor code is not assigned to any organization and so should not conflict with any real hardware.

It is always up to the user to determine that using these MAC addresses is okay on your network and won't conflict with anything else (including another simulation using Emu devices) on your network. If you are using the emulated net device in separate simulations you must consider global MAC address assignment issues and ensure that MAC addresses are unique across all simulations. The emulated net device respects the MAC address provided in the SetAddress method so you can do this manually. For larger simulations, you may want to set the OUI in the MAC address allocation function.

IP addresses corresponding to the emulated net devices are the addresses generated in the simulation, which are generated in the usual way via helper functions. Since we are using MAC spoofing, there will not be a conflict between ns-3 network stacks and any native network stacks.

The emulated net device comes with a helper function as all ns-3 devices do. One unique aspect is that there is no channel associated with the underlying medium. We really have no idea what this external medium is, and so have not made an effort to model it abstractly. The primary thing to be aware of is the implication this has for static global routing. The global router module attempts to walk the channels looking for adjacent networks. Since there is no channel, the global router will be unable to do this and you must then use a dynamic routing protocol such as OLSR to include routing in Emu-based networks.

6.1.2 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

6.2 Usage

6.2.1 Emu Net Device

The usage of the Emu net device is straightforward once the network of simulations has been configured. Since most of the work involved in working with this device is in network configuration before even starting a simulation, you may want to take a moment to review a couple of HOWTO pages on the ns-3 wiki that describe how to set up a virtual test network using VMware and how to run a set of example (client server) simulations that use Emu net devices.

http://www.nsnam.org/wiki/index.php/HOWTO_use_VMware_to_set_up_virtual_networks_(Windows) http://www.nsnam.org/wiki/index.php/HOWTO_use_ns-3_scripts_to_drive_real_hardware_(experimental)

Once you are over the configuration hurdle, the script changes required to use an `Emu` device are trivial. The main structural difference is that you will need to create an ns-3 simulation script for each node. In the case of the HOWTOs above, there is one client script and one server script. The only "challenge" is to get the addresses set correctly.

Just as with all other ns-3 net devices, we provide a helper class for the `Emu` net device. The following code snippet illustrates how one would declare an EmuHelper and use it to set the "DeviceName" attribute to "eth1" and install `Emu` devices on a group of nodes. You would do this on both the client and server side in the case of the HOWTO seen above.

```
EmuHelper emu;
emu.SetAttribute ("DeviceName", StringValue ("eth1"));
NetDeviceContainer d = emu.Install (n);
```

The only other change that may be required is to make sure that the address spaces (MAC and IP) on the client and server simulations are compatible. First the MAC address is set to a unique well-known value in both places (illustrated here for one side).

```
//
// We've got the devices in place.  Since we're using MAC address
// spoofing under the sheets, we need to make sure that the MAC addresses
// we have assigned to our devices are unique.  Ns-3 will happily
// automatically assign the same MAC addresses to the devices in both halves
// of our two-script pair, so let's go ahead and just manually change them
// to something we ensure is unique.
//
Ptr<NetDevice> nd = d.Get (0);
Ptr<EmuNetDevice> ed = nd->GetObject<EmuNetDevice> ();
ed->SetAddress ("00:00:00:00:00:02");
```

And then the IP address of the client or serveris set in the usual way using helpers.

```
//
// We've got the "hardware" in place.  Now we need to add IP addresses.
// This is the server half of a two-script pair.  We need to make sure
// that the addressing in both of these applications is consistent, so
// we use provide an initial address in both cases.  Here, the client
// will reside on one machine running ns-3 with one node having ns-3
// with IP address "10.1.1.2" and talk to a server script running in
// another ns-3 on another computer that has an ns-3 node with IP
// address "10.1.1.3"
//
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0", "0.0.0.2");
Ipv4InterfaceContainer i = ipv4.Assign (d);
```

You will use application helpers to generate traffic exactly as you do in any ns-3 simulation script. Note that the server address shown below in a snippet from the client, must correspond to the IP address assigned to the server node similarly to the snippet above.

```
uint32_t packetSize = 1024;
uint32_t maxPacketCount = 2000;
Time interPacketInterval = Seconds (0.001);
UdpEchoClientHelper client ("10.1.1.3", 9);
client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
client.SetAttribute ("Interval", TimeValue (interPacketInterval));
client.SetAttribute ("PacketSize", UintegerValue (packetSize));
ApplicationContainer apps = client.Install (n.Get (0));
apps.Start (Seconds (1.0));
apps.Stop (Seconds (2.0));
```

The `Emu` net device and helper provide access to ASCII and pcap tracing functionality just as other ns-3 net devices to. You enable tracing similarly to these other net devices:

```
EmuHelper::EnablePcapAll ("emu-udp-echo-client");
```

To see an example of a client script using the `Emu` net device, see `examples/emu-udp-echo-client.cc` and `examples/emu-udp-echo-server.cc` in the repository http://code.nsnam.org/craigdo/ns-3-emu/.

6.2.2 Tap Net Device

The `Tap` Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the `Tap` device is merged.

6.3 Implementation

Perhaps the most unusual part of the `Emu` and `Tap` device implementation relates to the requirement for executing some of the code with super-user permissions. Rather than force the user to execute the entire simulation as root, we provide a small "creator" program that runs as root and does any required high-permission sockets work.

We do a similar thing for both the `Emu` and the `Tap` devices. The high-level view is that the `CreateSocket` method creates a local interprocess (Unix) socket, forks, and executes the small creation program. The small program, which runs as suid root, creates a raw socket and sends back the raw socket file descriptor over the Unix socket that is passed to it as a parameter. The raw socket is passed as a control message (sometimes called ancillary data) of type SCM_RIGHTS.

6.3.1 Emu Net Device

The `Emu` net device uses the ns-3 threading and multithreaded real-time scheduler extensions. The interesting work in the `Emu` device is done when the net device is started (`EmuNetDevice::StartDevice ()`). An attribute ("Start") provides a simulation time at which to spin up the net device. At this specified time (which defaults to t=0), the socket creation function is called and executes as described above. You may also specify a time at which to stop the device using the "Stop" attribute.

Once the (promiscuous mode) socket is created, we bind it to an interface name also provided as an attribute ("DeviceName") that is stored internally as `m_deviceName`:

```
struct ifreq ifr;
bzero (&ifr, sizeof(ifr));
strncpy ((char *)ifr.ifr_name, m_deviceName.c_str (), IFNAMSIZ);
```

```
int32_t rc = ioctl (m_sock, SIOCGIFINDEX, &ifr);

struct sockaddr_ll ll;
bzero (&ll, sizeof(ll));

ll.sll_family = AF_PACKET;
ll.sll_ifindex = m_sll_ifindex;
ll.sll_protocol = htons(ETH_P_ALL);

rc = bind (m_sock, (struct sockaddr *)&ll, sizeof (ll));
```

After the promiscuous raw socket is set up, a separate thread is spawned to do reads from that socket and the link state is set to `Up`.

```
m_readThread = Create<SystemThread> (
  MakeCallback (&EmuNetDevice::ReadThread, this));
m_readThread->Start ();

NotifyLinkUp ();
```

The `EmuNetDevice::ReadThread` function basically just sits in an infinite loop reading from the promiscuous mode raw socket and scheduling packet receptions using the real-time simulator extensions.

```
for (;;)
  {
    ...

    len = recvfrom (m_sock, buf, bufferSize, 0, (struct sockaddr *)&addr,
      &addrSize);

    ...

    DynamicCast<RealtimeSimulatorImpl> (Simulator::GetImplementation ())->
      ScheduleRealtimeNow (
        MakeEvent (&EmuNetDevice::ForwardUp, this, buf, len));

    ...
  }
```

The line starting with our templated DynamicCast function probably deserves a comment. It gains access to the simulator implementation object using the `Simulator::GetImplementation` method and then casts to the real-time simulator implementation to use the real-time schedule method `ScheduleRealtimeNow`. This function will cause a handler for the newly received packet to be scheduled for execution at the current real time clock value. This will, in turn cause the simulation clock to be advanced to that real time value when the scheduled event (`EmuNetDevice::ForwardUp`) is fired.

The `ForwardUp` function operates as most other similar ns-3 net device methods do. The packet is first filtered based on the destination address. In the case of the `Emu` device, the

MAC destination address will be the address of the `Emu` device and not the hardware address of the real device. Headers are then stripped off and the trace hooks are hit. Finally, the packet is passed up the ns-3 protocol stack using the receive callback function of the net device.

Sending a packet is equally straightforward as shown below. The first thing we do is to add the ethernet header and trailer to the ns-3 `Packet` we are sending. The source address corresponds to the address of the `Emu` device and not the underlying native device MAC address. This is where the MAC address spoofing is done. The trailer is added and we enqueue and dequeue the packet from the net device queue to hit the trace hooks.

```
header.SetSource (source);
header.SetDestination (destination);
header.SetLengthType (packet->GetSize ());
packet->AddHeader (header);

EthernetTrailer trailer;
trailer.CalcFcs (packet);
packet->AddTrailer (trailer);

m_queue->Enqueue (packet);
packet = m_queue->Dequeue ();

struct sockaddr_ll ll;
bzero (&ll, sizeof (ll));

ll.sll_family = AF_PACKET;
ll.sll_ifindex = m_sll_ifindex;
ll.sll_protocol = htons(ETH_P_ALL);

rc = sendto (m_sock, packet->PeekData (), packet->GetSize (), 0,
  reinterpret_cast<struct sockaddr *> (&ll), sizeof (ll));
```

Finally, we simply send the packet to the raw socket which puts it out on the real network.

6.3.2 Tap Net Device

The `Tap` Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the `Tap` device is merged.

7 Packets

The design of the Packet framework of *ns* 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 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* 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.

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.

7.1 Packet design overview

Unlike *ns-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.

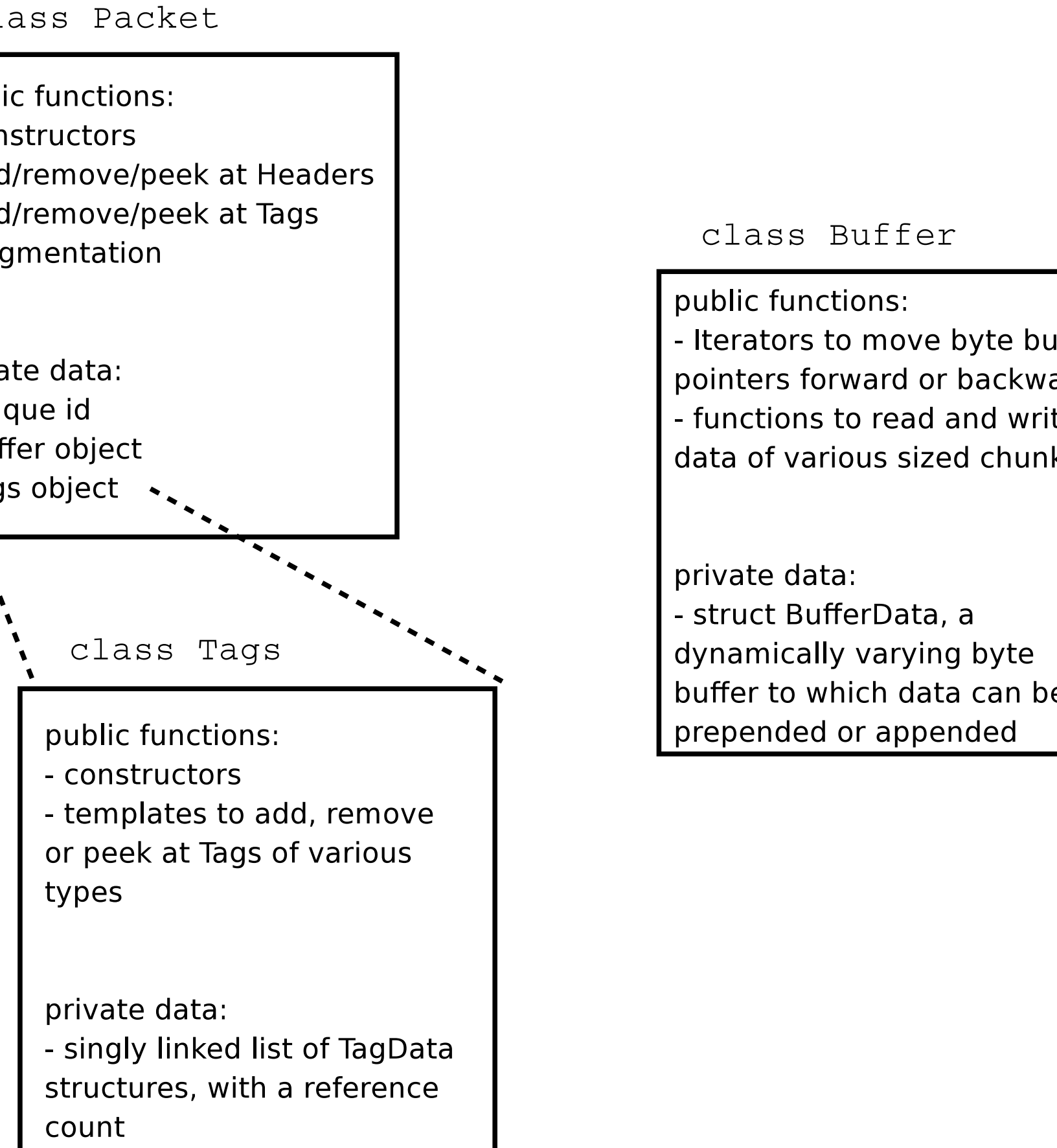

lass Packet
ic functions:
nstructors
d/remove/peek at Headers
d/remove/peek at Tags
gmentation
ate data:
que id
ffer object
gs object
class Tags
public functions:
- constructors
- templates to add, remove
or peek at Tags of various
types
private data:
- singly linked list of TagData
structures, with a reference
count
class Buffer
public functions:
- Iterators to move byte bu
pointers forward or backwa
- functions to read and writ
data of various sized chunk
private data:
- struct BufferData, a
dynamically varying byte
buffer to which data can be
prepended or appended

Figure Figure 7.1 is a high-level overview of the Packet implementation; more detail on the byte Buffer implementation is provided later in Figure Figure 7.2. In \nsthree, 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.

7.2 Packet interface

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

7.2.1 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);
```

7.2.2 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.
 */
```

7.2.3 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);
```

7.2.4 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);
```

7.2.5 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;
```

7.3 Using Headers

walk through an example of adding a UDP header

7.4 Using Tags

walk through an example of adding a flow ID

7.5 Using Fragmentation

walk through an example of link-layer fragmentation/reassembly

7.6 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;
}
```

7.7 Implementation details

7.7.1 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.

7.7.2 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

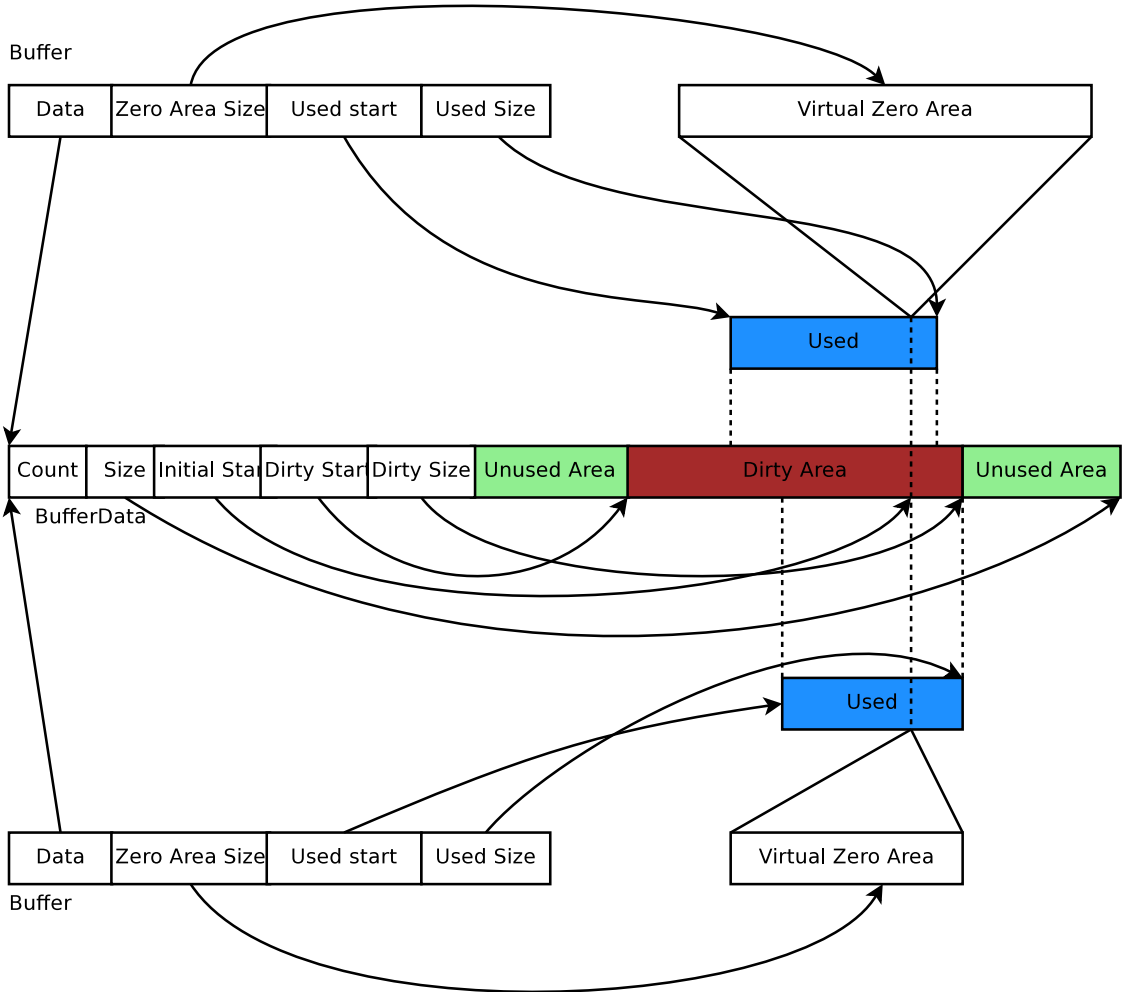


Figure 7.2: Implementation overview of a packet's byte Buffer.

This data structure is summarized in Figure Figure 7.2. 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.

7.7.3 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

7.7.4 Memory management

Describe free list.

Describe dataless vs. data-full packets.

7.7.5 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.

8 Sockets APIs

The sockets API is a long-standing API used by user-space applications to access network services in the kernel. A "socket" is an abstraction, like a Unix file handle, that allows applications to connect to other Internet hosts and exchange reliable byte streams and unreliable datagrams, among other services.

ns-3 provides two types of sockets APIs, and it is important to understand the differences between them. The first is a *native* ns-3 API, while the second uses the services of the native API to provide a POSIX-like API as part of an overall application process. Both APIs strive to be close to the typical sockets API that application writers on Unix systems are accustomed to, but the POSIX variant is much closer to a real system's sockets API.

8.1 ns-3 sockets API

The native sockets API for ns-3 provides an interface to various types of transport protocols (TCP, UDP) as well as to packet sockets and, in the future, Netlink-like sockets. However, users are cautioned to understand that the semantics are **not** the exact same as one finds in a real system (for an API which is very much aligned to real systems, see the next section).

`class ns3::Socket` is defined in `src/node/socket.cc,h`. Readers will note that many public member functions are aligned with real sockets function calls, and all other things being equal, we have tried to align with a Posix sockets API. However, note that:

- ns-3 applications handle a smart pointer to a Socket object, not a file descriptor;
- there is no notion of synchronous API or a "blocking" API; in fact, the model for interaction between application and socket is one of asynchronous I/O, which is not typically found in real systems (more on this below);
- the C-style socket address structures are not used;
- the API is not a complete sockets API, such as supporting all socket options or all function variants;
- many calls use `ns3::Packet` class to transfer data between application and socket. This may seem a little funny to people to pass "Packets" across a stream socket API, but think of these packets as just fancy byte buffers at this level (more on this also below).

8.1.1 Basic operation and calls

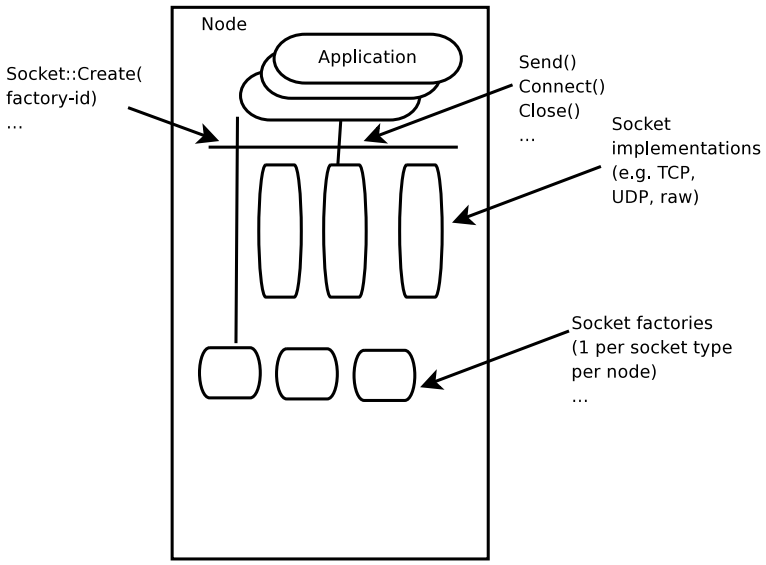


Figure 8.1: Implementation overview of native sockets API

8.1.1.1 Creating sockets

An application that wants to use sockets must first create one. On real systems, this is accomplished by calling socket():

```
int
socket(int domain, int type, int protocol);
```

which creates a socket in the system and returns an integer descriptor.

In ns-3, we have no equivalent of a system call at the lower layers, so we adopt the following model. There are certain *factory* objects that can create sockets. Each factory is capable of creating one type of socket, and if sockets of a particular type are able to be created on a given node, then a factory that can create such sockets must be aggregated to the Node.

```
static Ptr<Socket> CreateSocket (Ptr<Node> node, TypeId tid);
```

Examples of TypeIds to pass to this method are `TcpSocketFactory`, `PacketSocketFactory`, and `UdpSocketFactory`.

This method returns a smart pointer to a Socket object. Here is an example:

```
Ptr<Node> n0;
// Do some stuff to build up the Node's internet stack
Ptr<Socket> localSocket = Socket::CreateSocket (n0, TcpSocketFactory::GetTypeId ());
```

In some ns-3 code, sockets will not be explicitly created by user's main programs, if an ns-3 application does it. For instance, for `class ns3::OnOffApplication`, the function

`StartApplication()` performs the socket creation, and the application holds the socket pointer.

8.1.1.2 Using sockets

Below is a typical sequence of socket calls for a TCP client in a real implementation:

- `sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);`
- `bind(sock, ...);`
- `connect(sock, ...);`
- `send(sock, ...);`
- `recv(sock, ...);`
- `close(sock);`

There are analogs to all of these calls in ns-3, but we will focus on two aspects here. First, most usage of sockets in real systems requires a way to manage I/O between the application and kernel. These models include *blocking sockets*, *signal-based I/O*, and *non-blocking sockets* with polling. In ns-3, we make use of the callback mechanisms to support a fourth mode, which is analogous to POSIX *asynchronous I/O*.

In this model, on the sending side, if the `send()` call were to fail because of insufficient buffers, the application suspends the sending of more data until a function registered at the `SetSendCallback()` callback is invoked. An application can also ask the socket how much space is available by calling `GetTxAvailable ()`. A typical sequence of events for sending data (ignoring connection setup) might be:

- `SetSendCallback (MakeCallback(&HandleSendCallback));`
- `Send ();`
- `Send ();`
- ...
- `// Send fails because buffer is full`
- (wait until HandleSendCallback() is called)
- (HandleSendCallback() is called by socket, since space now available)
- `Send (); // Start sending again`

Similarly, on the receive side, the socket user does not block on a call to `recv()`. Instead, the application sets a callback with `SetRecvCallback ()` in which the socket will notify the application when (and how much) there is data to be read, and the application then calls `Recv()` to read the data until no more can be read.

8.1.2 Packet vs. buffer variants

There are two basic variants of `Send()` and `Recv()` supported:

```
virtual int Send (Ptr<Packet> p) = 0;
int Send (const uint8_t* buf, uint32_t size);

Ptr<Packet> Recv (void);
int Recv (uint8_t* buf, uint32_t size);
```

The non-Packet variants are left for legacy API reasons. When calling the raw buffer variant of `Send()`, the buffer is immediately written into a Packet and the `Send (Ptr<Packet> p)` is invoked.

Users may find it semantically odd to pass a Packet to a stream socket such as TCP. However, do not let the name bother you; think of `ns3::Packet` to be a fancy byte buffer. There are a few reasons why the Packet variants are more likely to be preferred in ns-3:

- Users can use the Tags facility of packets to, for example, encode a flow ID or other helper data.
- Users can exploit the copy-on-write implementation to avoid memory copies (on the receive side, the conversion back to a `uint8_t* buf` may sometimes incur an additional copy).
- Use of Packet is more aligned with the rest of the ns-3 API

8.1.3 Sending dummy data

Sometimes, users want the simulator to just pretend that there is an actual data payload in the packet (e.g. to calculate transmission delay) but do not want to actually produce or consume the data. This is straightforward to support in ns-3; have applications call `Create<Packet> (size);` instead of `Create<Packet> (buffer, size);`. Similarly, passing in a zero to the pointer argument in the raw buffer variants has the same effect. Note that, if some subsequent code tries to read the Packet data buffer, the fake buffer will be converted to a real (zero'ed) buffer on the spot, and the efficiency will be lost there.

8.1.4 Socket options

to be completed

8.1.5 Socket errno

to be completed

8.1.6 Example programs

to be completed

8.2 POSIX-like sockets API

this capability is under development and is scheduled for inclusion in the ns-3.5 releasetimeframe; see the repository http://code.nsnam.org/mathieu/ns-3-simu for details

The below is excerpted from Mathieu's post to ns-developers list on April 4, 2008.

"To summarize, the goal is that the full posix/socket API is defined in src/process/simu.h: each posix type and function is re-defined there with a simu_ or SIMU_ prefix to avoid ugly name clashes and collisions (feel free to come up with a better prefix).

Each process is created with a call to ProcessManager::Create and is attached to that ProcessManager instance. So, if the ProcessManager (which is aggregated to a Node in src/helper/process-helper.cc) is killed when the simulation ends, the system will automatically reclaim all the resources of each process associated to each manager. The same happens when an application "exits" from its main function.

The example application defines two posix "processes": the function ClientProgram creates a udp socket on the localhost port 2000 and the function ServerProgram creates a udp socket on the localhost port 2000. The code does not work right now because I did not get the details of simu_read right yet but, I do plan to make this work at some point.

I really think that this approach is worthwhile for many reasons, a few of which are outlined below:

- makes porting real world application code _much_ easier
- makes write applications for new users much easier because they can read the bsd socket api reference and documentation and write code directly.
- can be used to write applications which work in both simulation and in the real world at the same time. To do this, all you have to do is write your application to use the simu_ API, and, then, you can chose at compile-time which implementation of that API you want to use: you can pick one implementation which forwards all calls to the system BSD socket API or another one which forwards all calls to the attached ProcessManager. Arguably, I did not implement the version which forwards to system BSD sockets but, that should be pretty trivial.

So, anyway, comments about the overall API would be welcome. Students interested in the gsoc project for real-world code integration should consider looking at this also."

9 Node and Internet Stack

This chapter describes how ns-3 nodes are put together, and provides a walk-through of how packets traverse an internet-based Node.

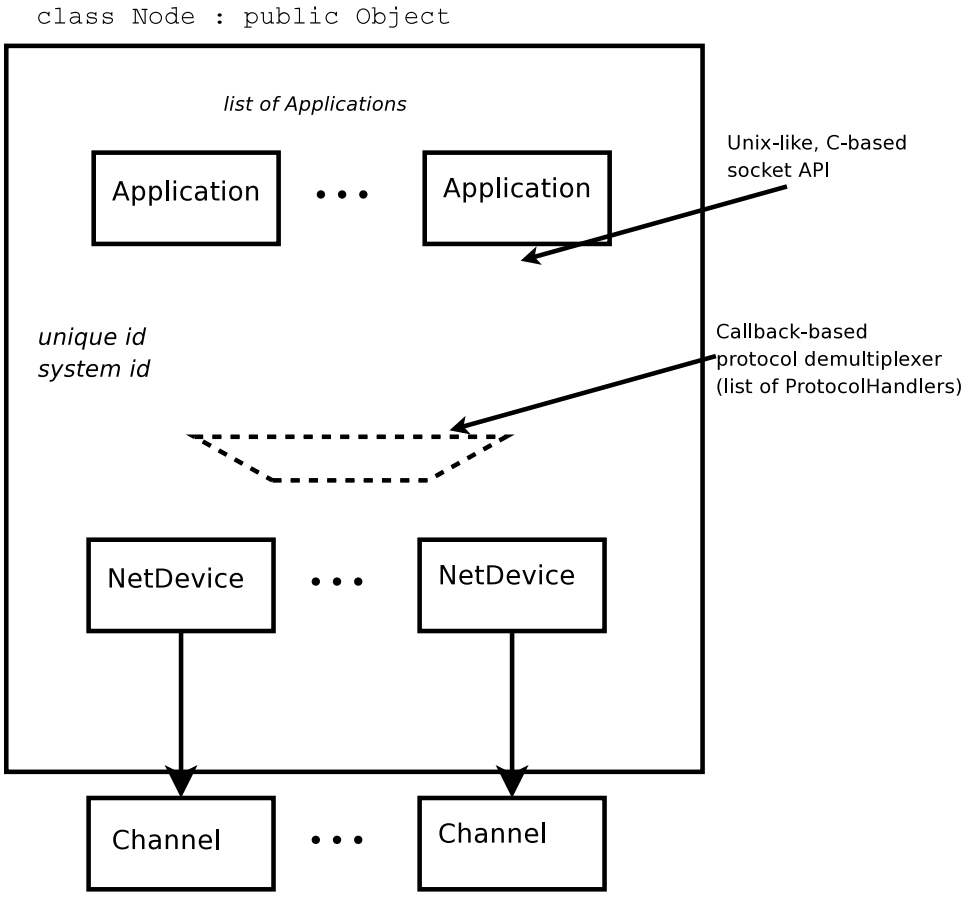


Figure 9.1: High-level node architecture.

In ns-3, nodes are instances of `class Node`. This class may be subclassed, but instead, the conceptual model is that we *aggregate* or insert objects to it rather than define subclasses.

One might think of a bare ns-3 node as a shell of a computer, to which one may add NetDevices (cards) and other innards including the protocols and applications. Figure 9.1 illustrates that Node objects contain a list of Applications (initially, the list is empty), a list of NetDevices (initially, the list is empty), a unique integer ID, and a system ID (for distributed simulation).

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

From a software perspective, the lower interface of applications corresponds to the C-based sockets API. The upper interface of NetDevice objects corresponds to the device independent sublayer of the Linux stack. Everything in between can be aggregated and plumbed together as needed.

Let's look more closely at the protocol demultiplexer. We want incoming frames at layer-2 to be delivered to the right layer-3 protocol such as Ipv4. The function of this demultiplexer is to register callbacks for receiving packets. The callbacks are indexed based on the EtherType in the layer-2 frame.

Many different types of higher-layer protocols may be connected to the NetDevice, such as IPv4, IPv6, ARP, MPLS, IEEE 802.1x, and packet sockets. Therefore, the use of a callback-based demultiplexer avoids the need to use a common base class for all of these protocols, which is problematic because of the different types of objects (including packet sockets) expected to be registered there.

Each NetDevice delivers packets to a callback with the following signature:

```
/**
 * \param device a pointer to the net device which is calling this callback
 * \param packet the packet received
 * \param protocol the 16 bit protocol number associated with this packet.
 *        This protocol number is expected to be the same protocol number
 *        given to the Send method by the user on the sender side.
 * \param address the address of the sender
 * \returns true if the callback could handle the packet successfully,
 *        false otherwise.
 */
typedef Callback<bool, Ptr<NetDevice>, Ptr<Packet>, uint16_t,
  const Address &> ReceiveCallback;
```

There is a function in class Node that matches that signature:

```
private:
  bool ReceiveFromDevice (Ptr<NetDevice> device, Ptr<Packet>,
                          uint16_t protocol, const Address &from);
```

However, users do not need to access this function directly. Instead, when users call `uint32_t AddDevice (Ptr<NetDevice> device)`, the implementation of this function sets the callback (and the function returns the ifIndex of the NetDevice on that Node).

But what does the ReceiveFromDevice function do? Here, it looks up another callback, in its list of callbacks, corresponding to the matching EtherType. This callback is called a ProtocolHandler, and is specified as follows:

```
typedef Callback<void, Ptr<NetDevice>, Ptr<Packet>, uint16_t,
  const Address &> ProtocolHandler;
```

Upper-layer protocols or objects are expected to provide such a function. and register it with the list of ProtocolHandlers by calling `Node::RegisterProtocolHandler ();` For instance, if Ipv4 is aggregated to a Node, then the Ipv4 receive function can be registered with the protocol handler by calling:

```
RegisterProtocolHandler (
  MakeCallback (&Ipv4L3Protocol::Receive, ipv4),
```

```
Ipv4L3Protocol::PROT_NUMBER, 0);
```

and likewise for Ipv6, Arp, etc.

9.1 NodeList

Every Node created is automatically added to the ns-3 `NodeList`. The NodeList class provides an `Add()` method and C++ iterators to allow one to walk the node list or fetch a Node pointer by its integer identifier.

9.2 Internet stack aggregation

The above `class Node` is not very useful as-is; other objects must be aggregated to it to provide useful node functionality.

The ns-3 source code directory `src/internet-stack` provides implmentation of TCP/IPv4-related components. These include IPv4, ARP, UDP, TCP, and other related protocols.

Internet Nodes are not subclasses of class Node; they are simply Nodes that have had a bunch of IPv4-related objects aggregated to them. They can be put together by hand, or via a helper function `AddInternetStack ()` which does the following:

```
void AddInternetStack (Ptr<Node> node)
{
  // Create layer-3 protocols
  Ptr<Ipv4L3Protocol> ipv4 = CreateObject<Ipv4L3Protocol> ();
  Ptr<ArpL3Protocol> arp = CreateObject<ArpL3Protocol> ();
  ipv4->SetNode (node);
  arp->SetNode (node);

  // Create an L4 demux
  Ptr<Ipv4L4Demux> ipv4L4Demux = CreateObject<Ipv4L4Demux> ();

  // Create transport protocols and insert them into the demux
  Ptr<UdpL4Protocol> udp = CreateObject<UdpL4Protocol> ();
  Ptr<TcpL4Protocol> tcp = CreateObject<TcpL4Protocol> ();

  ipv4L4Demux->SetNode (node);
  udp->SetNode (node);
  tcp->SetNode (node);

  ipv4L4Demux->Insert (udp);
  ipv4L4Demux->Insert (tcp);

  // Add factories for instantiating transport protocol sockets
  Ptr<UdpSocketFactoryImpl> udpFactory = CreateObject<UdpSocketFactoryImpl> ();
  Ptr<TcpSocketFactoryImpl> tcpFactory = CreateObject<TcpSocketFactoryImpl> ();
  Ptr<Ipv4Impl> ipv4Impl = CreateObject<Ipv4Impl> ();

  udpFactory->SetUdp (udp);
```

```
  tcpFactory->SetTcp (tcp);
  ipv4Impl->SetIpv4 (ipv4);

  // Aggregate all of these new objects to the node
  node->AggregateObject (ipv4);
  node->AggregateObject (arp);
  node->AggregateObject (ipv4Impl);
  node->AggregateObject (udpFactory);
  node->AggregateObject (tcpFactory);
  node->AggregateObject (ipv4L4Demux);
}
```

9.2.1 Internet Node structure

The Internet Node (an ns-3 Node augmented by aggregation to have one or more IP stacks) has the following internal structure.

9.2.1.1 Layer-3 protocols

At the lowest layer, sitting above the NetDevices, are the "layer 3" protocols, including IPv4, IPv6, and ARP. These protocols provide the following key methods and data members:

```
class Ipv4L3Protocol : public Object
{
public:
  // Add an Ipv4 interface corresponding to the provided NetDevice
  uint32_t AddInterface (Ptr<NetDevice> device);

  // Receive function that can be bound to a callback, for receiving
  // packets up the stack
  void Receive( Ptr<NetDevice> device, Ptr<Packet> p, uint16_t protocol,
    const Address &from);

  // Higher-level layers call this method to send a packet
  // down the stack to the MAC and PHY layers
  //
  void Send (Ptr<Packet> packet, Ipv4Address source,
             Ipv4Address destination, uint8_t protocol);

private:
  Ipv4InterfaceList m_interfaces;

  // Protocol handlers
}
```

There are many more functions (such as `Forward ()`) but we will focus on the above four items from an architectural perspective.

First, note that the `Receive ()` function has a matching signature to the ReceiveCallback in the `class Node`. This function pointer is inserted into the the Node's protocol handler

when `AddInterface ()` is called. The actual registration is done with a statement such as: follows:

```
RegisterProtocolHandler ( MakeCallback (&Ipv4Protocol::Receive, ipv4),
   Ipv4L3Protocol::PROT_NUMBER, 0);
```

The Ipv4L3Protocol object is aggregated to the Node; there is only one such Ipv4L3Protocol object. Higher-layer protocols that have a packet to send down to the Ipv4L3Protocol object can call `GetObject<Ipv4L3Protocol> ()` to obtain a pointer, as follows:

```
Ptr<Ipv4L3Protocol> ipv4 = m_node->GetObject<Ipv4L3Protocol> ();
if (ipv4 != 0)
  {
    ipv4->Send (packet, saddr, daddr, PROT_NUMBER);
  }
```

This class nicely demonstrates two techniques we exploit in ns-3 to bind objects together: callbacks, and object aggregation.

Once IPv4 has determined that a packet is for the local node, it forwards it up the stack. This is done with the following function:

```
void
Ipv4L3Protocol::ForwardUp (Ptr<Packet> p, Ipv4Header const&ip,
                           Ptr<Ipv4Interface> incomingInterface)
{
  NS_LOG_FUNCTION (this << p << &ip);

  Ptr<Ipv4L4Demux> demux = m_node->GetObject<Ipv4L4Demux> ();
  Ptr<Ipv4L4Protocol> protocol = demux->GetProtocol (ip.GetProtocol ());
  protocol->Receive (p, ip.GetSource (), ip.GetDestination (), incomingInterface);
}
```

The first step is to find the aggregated Ipv4L4Demux object. Then, this object is consulted to look up the right Ipv4L4Protocol, based on IP protocol number. For instance, TCP is registered in the demux as protocol number 6. Finally, the `Receive()` function on the Ipv4L4Protocol (such as `TcpL4Protocol::Receive` is called.

We have not yet introduced the class Ipv4Interface. Basically, each NetDevice is paired with an IPv4 representation of such device. In Linux, this `class Ipv4Interface` roughly corresponds to the `struct in_device`; the main purpose is to provide address-family specific information (addresses) about an interface.

9.2.1.2 Layer-4 protocols and sockets

We next describe how the transport protocols, sockets, and applications tie together. In summary, each transport protocol implementation is a socket factory. An application that needs a new socket

For instance, to create a UDP socket, an application would use a code snippet such as the following:

```
Ptr<Udp> udpSocketFactory = GetNode ()->GetObject<Udp> ();
Ptr<Socket> m_socket = socketFactory->CreateSocket ();
```

```
m_socket->Bind (m_local_address);
...
```

The above will query the node to get a pointer to its UDP socket factory, will create one such socket, and will use the socket with an API similar to the C-based sockets API, such as `Connect ()` and `Send ()`. See the chapter on ns-3 sockets for more information.

We have described so far a socket factory (e.g. `class Udp`) and a socket, which may be specialized (e.g., `class UdpSocket`). There are a few more key objects that relate to the specialized task of demultiplexing a packet to one or more receiving sockets. The key object in this task is `class Ipv4EndPointDemux`. This demultiplexer stores objects of `class Ipv4EndPoint`. This class holds the addressing/port tuple (local port, local address, destination port, destination address) associated with the socket, and a receive callback. This receive callback has a receive function registered by the socket. The `Lookup ()` function to Ipv4EndPointDemux returns a list of Ipv4EndPoint objects (there may be a list since more than one socket may match the packet). The layer-4 protocol copies the packet to each Ipv4EndPoint and calls its `ForwardUp ()` method, which then calls the `Receive ()` function registered by the socket.

An issue that arises when working with the sockets API on real systems is the need to manage the reading from a socket, using some type of I/O (e.g., blocking, non-blocking, asynchronous, ...). ns-3 implements an asynchronous model for socket I/O; the application sets a callback to be notified of received data ready to be read, and the callback is invoked by the transport protocol when data is available. This callback is specified as follows:

```
void Socket::SetRecvCallback (Callback<void, Ptr<Socket>,
  Ptr<Packet>, const Address&> receivedData);
```

The data being received is conveyed in the Packet data buffer. An example usage is in `class PacketSink`:

```
m_socket->SetRecvCallback (MakeCallback(&PacketSink::HandleRead, this));
```

To summarize, internally, the UDP implementation is organized as follows:

- a `UdpImpl` class that implements the Udp socket factory functionality
- a `UdpL4Protocol` class that implements the protocol logic that is socket-independent
- a `UdpSocketImpl` class that implements socket-specific aspects of UDP
- a class called `Ipv4EndPoint` that stores the addressing tuple (local port, local address, destination port, destination address) associated with the socket, and a receive callback for the socket.

9.2.2 Internet Node interfaces

Many of the implementation details, or internal objects themselves, of Internet Node objects are not exposed at the simulator public API. This allows for different implementations; for instance, replacing the native ns-3 models with ported TCP/IP stack code.

The C++ public APIs of all of these objects is found in the `src/node` directory, including principally:

- `socket.h`
- `tcp.h`
- `udp.h`

- `ipv4.h`

These are typically base class objects that implement the default values used in the implementation, implement access methods to get/set state variables, host attributes, and implement publicly-available methods exposed to clients such as `CreateSocket`.

9.2.3 Example path of a packet

These two figures show an example stack trace of how packets flow through the Internet Node objects.

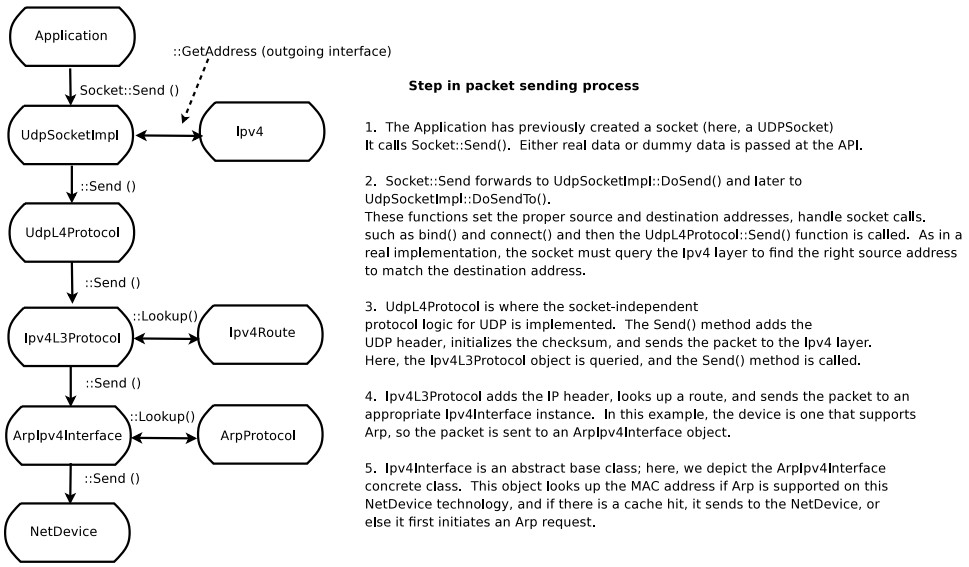
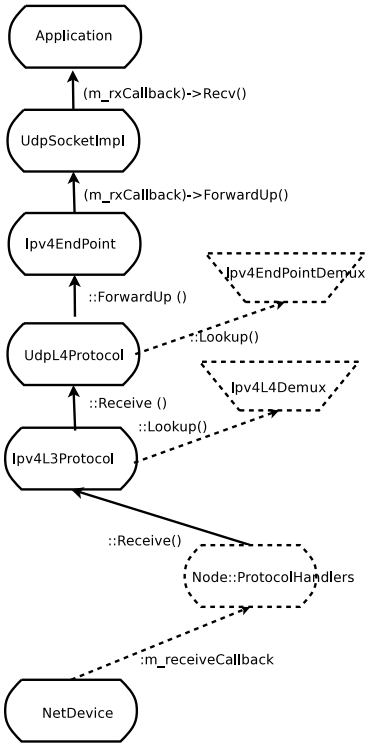


Figure 9.2: Send path of a packet.



Step in packet receiving process

7. UdpSocketImpl itself calls the Recv() callback set by the Application when data is ready to be read. The application can then call the socket Recv() or RecvFrom() methods to read data (or dummy data) from the socket.

6. Ipv4EndPoint has a callback where a Socket object is able to register a receive method. Here, this callback calls to UdpSocketImpl::ForwardUp()

5. UdpL4Protocol is where the socket-independent protocol logic for UDP is implemented. The Receive() method removes the UDP header and looks up the per-flow context state, which is one or more Ipv4EndPoint objects stored in an Ipv4EndPointDemux (indexed by src addr, src port, dest addr, dest port). It then calls Ipv4EndPoint::ForwardUp() when done.

4. Ipv4L3Protocol removes the IP header, checks checksum (if implemented), and either Forwards the packet or calls ForwardUp(). ForwardUp() then looks up the IP protocol number in an Ipv4L4Demux to obtain a pointer to an Ipv4L4Protocol and calls the ::Receive() method.

3. Node::ReceiveFromDevice stores a set of callbacks (protocol handlers) that are looked up based on protocol number and device. In this case, the lookup will result in an Ipv4L3Protocol::Receive() being called.

2. This is typically the Node::ReceiveFromDevice() function

1. NetDevice calls the function registered at Node::m_receiveCallback

Figure 9.3: Receive path of a packet.

10 TCP models in ns-3

This chapter describes the TCP models available in ns-3.

10.1 Generic support for TCP

ns-3 was written to support multiple TCP implementations. The implementations inherit from a few common header classes in the `src/node` directory, so that user code can swap out implementations with minimal changes to the scripts.

There are two important abstract base classes:

- `class TcpSocket`: This is defined in `src/node/tcp-socket.cc,h`. This class exists for hosting TcpSocket attributes that can be reused across different implementations. For instance, `TcpSocket::SetInitialCwnd()` can be used for any of the implementations that derive from `class TcpSocket`.
- `class TcpSocketFactory`: This is used by applications to create TCP sockets. A typical usage can be seen in this snippet:

```
// Create the socket if not already created
if (!m_socket)
  {
    m_socket = Socket::CreateSocket (GetNode(), m_tid);
    m_socket->Bind (m_local);
    ...
  }
```

  The parameter `m_tid` controls the TypeId of the actual Tcp Socket implementation that is instantiated. This way, the application can be written generically and different socket implementations can be swapped out by specifying the TypeId.

10.2 ns-3 TCP

ns-3 contains a port of the TCP model from GTNetS. This model is a full TCP, in that it is bidirectional and attempts to model the connection setup and close logic. In fact, it is a more complete implementation of the TCP state machine than ns-2's "FullTcp" model. This TCP model was originally written by George Riley as part of GTNetS and ported to ns-3 by Raj Bhattacharjea.

The implementation of TCP is contained in the following files:

```
src/internet-stack/tcp-header.{cc,h}
src/internet-stack/tcp-l4-protocol.{cc,h}
src/internet-stack/tcp-socket-factory-impl.{cc,h}
src/internet-stack/tcp-socket-impl.{cc,h}
src/internet-stack/tcp-typedefs.h
src/internet-stack/rtt-estimator.{cc,h}
src/internet-stack/sequence-number.{cc,h}
```

10.2.1 Usage

The file `examples/tcp-star-server.cc` contains an example that makes use of `ns3::OnOffApplication` and `ns3::PacketSink` applications.

Using the helper functions defined in `src/helper`, here is how one would create a TCP receiver:

```
// Create a packet sink on the star "hub" to receive these packets
uint16_t port = 50000;
Address sinkLocalAddress(InetSocketAddress (Ipv4Address::GetAny (), port));
PacketSinkHelper sinkHelper ("ns3::TcpSocketFactory", sinkLocalAddress);
ApplicationContainer sinkApp = sinkHelper.Install (serverNode);
sinkApp.Start (Seconds (1.0));
sinkApp.Stop (Seconds (10.0));
```

Similarly, the below snippet configures OnOffApplication traffic source to use TCP:

```
// Create the OnOff applications to send TCP to the server
OnOffHelper clientHelper ("ns3::TcpSocketFactory", Address ());
```

The careful reader will note above that we have specified the TypeId of an abstract base class `TcpSocketFactory`. How does the script tell ns-3 that it wants the native ns-3 TCP vs. some other one? Well, when internet stacks are added to the node, the default TCP implementation that is aggregated to the node is the ns-3 TCP. This can be overridden as we show below when using Network Simulation Cradle. So, by default, when using the ns-3 helper API, the TCP that is aggregated to nodes with an Internet stack is the native ns-3 TCP.

Once a TCP socket is created, you will want to follow conventional socket logic and either connect() and send() (for a TCP client) or bind(), listen(), and accept() (for a TCP server). See Chapter 8 [Sockets API], page 51 for a review of how sockets are used in ns-3.

To configure behavior of TCP, a number of parameters are exported through the Chapter 3 [ns-3 attribute system], page 12. These are documented in the Doxygen for `class TcpSocket`.

10.2.2 Current limitations

- Only Tahoe congestion control is presently supported.
- Only IPv4 is supported (IPv6 support will start to be added in ns-3.3).
- Bug 198: TcpSocketImpl doesn't send acks with data packets in two-way transfers
- Bug 250: Tcp breaks if you set the DelAckCount parameter to be greater than 2
- Bug 311: Tcp socket close returns -1 but does not set errno.

10.3 Network Simulation Cradle

The Network Simulation Cradle (NSC) is a framework for wrapping real-world network code into simulators, allowing simulation of real-world behavior at little extra cost. This work has been validated by comparing situations using a test network with the same situations in the simulator. To date, it has been shown that the NSC is able to produce extremely accurate results. NSC supports four real world stacks: FreeBSD, OpenBSD, lwIP and Linux. Emphasis has been placed on not changing any of the network stacks by hand. Not a single line of code has been changed in the network protocol implementations of any of the above four stacks. However, a custom C parser was built to programmatically change source code.

NSC has previously been ported to ns-2 and OMNeT++, and recently was added to ns-3. This section describes the ns-3 port of NSC and how to use it.

10.3.1 Prerequisites

Presently, NSC has been tested and shown to work on these platforms: Linux i386 and Linux x86-64. NSC does not support powerpc at the moment.

NSC requires the packages mercurial, flex, and bison.

10.3.2 Configuring and Downloading

NSC is disbled by default and must be explicitly configured in. To try this, type

```
./waf configure --enable-nsc
```

the output of the configuration will show something like:

```
Checking for NSC supported architecture x86_64                          : ok
Pulling nsc updates from https://secure.wand.net.nz/mercurial/nsc
pulling from https://secure.wand.net.nz/mercurial/nsc
searching for changes
no changes found
---- Summary of optional NS-3 features:
...
Network Simulation Cradle     : enabled
...
```

if successful. Note that the configure script pulls a recent copy of NSC from a mercurial repository. This download will not work if you are not online.

If everything went OK, you will see a directory called "nsc" in the top-level directory, with contents like this:

```
audit.sh            linux-2.6/          openbsd3/           scons-time.py*
ChangeLog           linux-2.6.18/       README              SConstruct
config.log          linux-2.6.26/       sconsign.py*        sim/
freebsd5/           lwip-1.3.0/         scons-LICENSE       test/
globaliser/         lwip-HEAD/          scons-local-1.0.1/
INSTALL             ns/                 scons.py*
LICENSE             omnetpp/            scons-README
```

10.3.3 Building and validating

Building ns-3 with nsc support is the same as building it without; no additional arguments are needed for waf. Building nsc may take some time compared to ns-3; it is interleaved in the ns-3 building process.

Try running the regression tests: `./waf --regression`. If NSC has been successfully built, the following test should show up in the results:

```
PASS test-tcp-nsc-lfn
```

This confirms that NSC is ready to use.

10.3.4 Usage

There are a few example files. Try

```
./waf --run tcp-nsc-zoo
./waf --run tcp-nsc-lfn
```

These examples will deposit some `.pcap` files in your directory, which can be examined by tcpdump or wireshark.

Let's look at the `examples/tcp-nsc-zoo.cc` file for some typical usage. How does it differ from using native ns-3 TCP? There is one main configuration line, when using NSC and the ns-3 helper API, that needs to be set:

```
InternetStackHelper internetStack;

internetStack.SetNscStack ("liblinux2.6.26.so");
// this switches nodes 0 and 1 to NSCs Linux 2.6.26 stack.
internetStack.Install (n.Get(0));
internetStack.Install (n.Get(1));
```

The key line is the `SetNscStack`. This tells the InternetStack helper to aggregate instances of NSC TCP instead of native ns-3 TCP to the remaining nodes. It is important that this function be called **before** callling the `Install()` function, as shown above.

Which stacks are available to use? Presently, the focus has been on Linux 2.6.18 and Linux 2.6.26 stacks for ns-3. To see which stacks were built, one can execute the following find command at the ns-3 top level directory:

```
~/ns-3.2> find nsc -name "*.so" -type f
nsc/linux-2.6.18/liblinux2.6.18.so
nsc/linux-2.6.26/liblinux2.6.26.so
```

This tells us that we may either pass the library name liblinux2.6.18.so or liblinux2.6.26.so to the above configuration step.

10.3.5 Stack configuration

NSC TCP shares the same configuration attributes that are common across TCP sockets, as described above and documented in Doxygen

Additionally, NSC TCP exports a lot of configuration variables into the ns-3 Chapter 3 [Attributes], page 12 system, via a sysctl-like interface. In the `examples/tcp-nsc-zoo` example, you can see the following configuration:

```
// this disables TCP SACK, wscale and timestamps on node 1 (the attributes represent sysc
Config::Set ("/NodeList/1/$ns3::Ns3NscStack<linux2.6.26>/net.ipv4.tcp_sack", StringValue
Config::Set ("/NodeList/1/$ns3::Ns3NscStack<linux2.6.26>/net.ipv4.tcp_timestamps", String
Config::Set ("/NodeList/1/$ns3::Ns3NscStack<linux2.6.26>/net.ipv4.tcp_window_scaling", St
```

These additional configuration variables are not available to native ns-3 TCP.

10.3.6 NSC API

This subsection describes the API that NSC presents to ns-3 or any other simulator. NSC provides its API in the form of a number of classes that are defined in `sim/sim_interface.h` in the nsc directory.

- **INetStack** INetStack contains the 'low level' operations for the operating system network stack, e.g. in and output functions from and to the network stack (think of this as the 'network driver interface'. There are also functions to create new TCP or UDP sockets.
- **ISendCallback** This is called by NSC when a packet should be sent out to the network. This simulator should use this callback to re-inject the packet into the simulator so the actual data can be delivered/routed to its destination, where it will eventually be handed into Receive() (and eventually back to the receivers NSC instance via INetStack->if_receive()).
- **INetStreamSocket** This is the structure defining a particular connection endpoint (file descriptor). It contains methods to operate on this endpoint, e.g. connect, disconnect, accept, listen, send_data/read_data, ...
- **IInterruptCallback** This contains the wakeup callback, which is called by NSC whenever something of interest happens. Think of wakeup() as a replacement of the operating systems wakeup function: Whenever the operating system would wake up a process that has been waiting for an operation to complete (for example the TCP handshake during connect()), NSC invokes the wakeup() callback to allow the simulator to check for state changes in its connection endpoints.

10.3.7 ns-3 implementation

The ns-3 implementation makes use of the above NSC API, and is implemented as follows.

The three main parts are:

- `ns3::NscTcpL4Protocol`: a subclass of Ipv4L4Protocol (and two nsc classes: ISendCallback and IInterruptCallback)
- `ns3::NscTcpSocketImpl`: a subclass of TcpSocket
- `ns3::NscTcpSocketFactoryImpl`: a factory to create new NSC sockets

`src/internet-stack/nsc-tcp-l4-protocol` is the main class. Upon Initialization, it loads an nsc network stack to use (via dlopen()). Each instance of this class may use a different stack. The stack (=shared library) to use is set using the SetNscLibrary() method (at this time its called indirectly via the internet stack helper). The nsc stack is then set up accordingly (timers etc). The NscTcpL4Protocol::Receive() function hands the packet it receives (must be a complete tcp/ip packet) to the nsc stack for further processing. To be able to send packets, this class implements the nsc send_callback method. This method is called by nsc whenever the nsc stack wishes to send a packet out to the network. Its arguments are a raw buffer, containing a complete TCP/IP packet, and a length value. This method therefore has to convert the raw data to a Ptr<Packet> usable by ns-3. In order to avoid various ipv4 header issues, the nsc ip header is not included. Instead, the tcp header and the actual payload are put into the Ptr<Packet>, after this the Packet is passed down to layer 3 for sending the packet out (no further special treatment is needed in the send code path).

This class calls `ns3::NscTcpSocketImpl` both from the nsc wakeup() callback and from the Receive path (to ensure that possibly queued data is scheduled for sending).

`src/internet-stack/nsc-tcp-socket-impl` implements the nsc socket interface. Each instance has its own nscTcpSocket. Data that is Send() will be handed to the nsc stack

via m_nscTcpSocket->send_data(). (and not to nsc-tcp-l4, this is the major difference compared to ns-3 TCP). The class also queues up data that is Send() before the underlying descriptor has entered an ESTABLISHED state. This class is called from the nsc-tcp-l4 class, when the nsc-tcp-l4 wakeup() callback is invoked by nsc. nsc-tcp-socket-impl then checks the current connection state (SYN_SENT, ESTABLISHED, LISTEN...) and schedules appropriate callbacks as needed, e.g. a LISTEN socket will schedule Accept to see if a new connection must be accepted, an ESTABLISHED socket schedules any pending data for writing, schedule a read callback, etc.

Note that `ns3::NscTcpSocketImpl` does not interact with nsc-tcp directly: instead, data is redirected to nsc. nsc-tcp calls the nsc-tcp-sockets of a node when its wakeup callback is invoked by nsc.

10.3.8 Limitations

- NSC only works on single-interface nodes; attempting to run it on a multi-interface node will cause a program error. This limitation should be fixed by ns-3.3.
- Cygwin and OS X PPC are not presently supported
- The non-Linux stacks of NSC are not supported
- NSC's integration into the build system presently requires on-line access and mercurial, and is a slow download.

For more information, see this wiki page.

11 Routing overview

This chapter describes the overall design of routing in the `src/internet-stack` module, and some details about the routing approachs currently implemented.

11.1 Overview

We intend to support traditional routing approaches and protocols, ports of open source routing implementations, and facilitate research into unorthodox routing techniques. For simulations that are not primarily focused on routing and that simply want correct routing tables to occur somehow, we have an global centralized routing capability. A singleton object (GlobalRouteManager) be instantiated, builds a network map, and populates a forwarding table on each node at time t=0 in the simulation. Simulation script writers can use the same node API to manually enter routes as well.

11.2 Support for multiple routing protocols

Typically, multiple routing protocols are supported in user space and coordinate to write a single forwarding table in the kernel. Presently in `ns-3`, the implementation instead allows for multiple routing protocols to build/keep their own routing state, and the IPv4 implementation will query each one of these routing protocols (in some order determined by the simulation author) until a route is found.

We chose this approach because it may better faciliate the integration of disparate routing approaches that may be difficult to coordinate the writing to a single table, approaches where more information than destination IP address (e.g., source routing) is used to determine the next hop, and on-demand routing approaches where packets must be cached.

There are presently two routing protocols defined:

- class Ipv4StaticRouting (covering both unicast and multicast)
- Optimized Link State Routing (a MANET protocol defined in RFC 3626)

but first we describe how multiple routing protocols are supported.

11.2.1 class Ipv4RoutingProtocol

`class Ipv4RoutingProtocol` derives from ns-3 Object which means that it supports interface aggregation and reference counting. Routing protocols should inherit from this class, defined in src/node/ipv4.cc.

The main function that must be supported by these protocols is called `RequestRoute`.

```
* This method is called whenever a node's IPv4 forwarding engine
* needs to lookup a route for a given packet and IP header.
*
* The routing protocol implementation may determine immediately it
* should not be handling this particular the route request.  For
* instance, a routing protocol may decline to search for routes for
* certain classes of addresses, like link-local.  In this case,
* RequestRoute() should return false and the routeReply callback
* must not be invoked.
*
```

```
   * If the routing protocol implementations assumes it can provide
   * the requested route, then it should return true, and the
   * routeReply callback must be invoked, either immediately before
   * returning true (synchronously), or in the future (asynchronous).
   * The routing protocol may use any information available in the IP
   * header and packet as routing key, although most routing protocols
   * use only the destination address (as given by
   * ipHeader.GetDestination ()).  The routing protocol is also
   * allowed to add a new header to the packet, which will appear
   * immediately after the IP header, although most routing do not
   * insert any extra header.
   */
  virtual bool RequestRoute (uint32_t ifIndex,
                             const Ipv4Header &ipHeader,
                             Ptr<Packet> packet,
                             RouteReplyCallback routeReply) = 0;
```

This class also provides a typedef (used above) for a special Callback that will pass to the callback function the Ipv4Route that is found (see the Doxygen documentation):

```
  typedef Callback<void, bool, const Ipv4Route&, Ptr<Packet>, const Ipv4Header&> RouteReply
```

11.2.2 Ipv4::AddRoutingProtocol

Class Ipv4 provides a pure virtual function declaration for the method that allows one to add a routing protocol:

```
  void AddRoutingProtocol (Ptr<Ipv4RoutingProtocol> routingProtocol,
                           int16_t priority);
```

This method is implemented by class Ipv4L3Protocol in the internet-stack module.

The priority variable above governs the priority in which the routing protocols are inserted. Notice that it is a signed int. When the class Ipv4L3Protocol is instantiated, a single routing protocol (Ipv4StaticRouting, introduced below) is added at priority zero. Internally, a list of Ipv4RoutingProtocols is stored, and and the routing protocols are each consulted in decreasing order of priority to see whether a match is found. Therefore, if you want your Ipv4RoutingProtocol to have priority lower than the static routing, insert it with priority less than 0; e.g.:

```
  m_ipv4->AddRoutingProtocol (m_routingTable, -10);
```

11.2.3 Ipv4L3Protocol::Lookup

The main function for obtaining a route is shown below:

```
Ipv4L3Protocol::Lookup (
  uint32_t ifIndex,
  Ipv4Header const &ipHeader,
  Ptr<Packet> packet,
  Ipv4RoutingProtocol::RouteReplyCallback routeReply)
```

This function will search the list of routing protocols, in priority order, until a route is found. It will then invoke the RouteReplyCallback and no further routing protocols will be

searched. If the caller does not want to constrain the possible interface, it can be wildcarded as such:

```
Lookup (Ipv4RoutingProtocol::IF_INDEX_ANY, ipHeader, packet, routeReply);
```

11.3 Roadmap and Future work

Some goals for future support are:

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
```

Global computation of multicast routing 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.

In the future, work on bringing XORP or quagga routing to ns, but it will take several months to port and enable.

There are presently no roadmap plans for IPv6.

11.4 Static routing

The internet-stack module provides one routing protocol (Ipv4StaticRouting) by default. This routing protocol allows one to add unicast or multicast static routes to a node.

11.5 Unicast routing

The unicast static routing API may be accessed via the functions

```
void Ipv4::AddHostRouteTo ()
void Ipv4::AddNetworkRouteTo ()
void Ipv4::SetDefaultRoute ()
uint32_t Ipv4::GetNRoutes ()
Ipv4Route Ipv4::GetRoute ()
```

Doxygen documentation provides full documentation of these methods. These methods are forwarding functions to the actual implementation in Ipv4StaticRouting, when using the internet-stack module.

11.6 Multicast routing

The following function is used to add a static multicast route to a node:

```
void
Ipv4StaticRouting::AddMulticastRoute (Ipv4Address origin,
                          Ipv4Address group,
                          uint32_t inputInterface,
                          std::vector<uint32_t> outputInterfaces);
```

A multicast route must specify an origin IP address, a multicast group and an input network interface index as conditions and provide a vector of output network interface indices over which packets matching the conditions are sent.

Typically there are two main types of multicast routes: routes of the first kind are used during forwarding. All of the conditions must be exlicitly provided. The second kind of routes are used to get packets off of a local node. The difference is in the input interface. Routes for forwarding will always have an explicit input interface specified. Routes off of a node will always set the input interface to a wildcard specified by the index Ipv4RoutingProtocol::IF_INDEX_ANY.

For routes off of a local node wildcards may be used in the origin and multicast group addresses. The wildcard used for Ipv4Adresses is that address returned by Ipv4Address::GetAny () – typically "0.0.0.0". Usage of a wildcard allows one to specify default behavior to varying degrees.

For example, making the origin address a wildcard, but leaving the multicast group specific allows one (in the case of a node with multiple interfaces) to create different routes using different output interfaces for each multicast group.

If the origin and multicast addresses are made wildcards, you have created essentially a default multicast address that can forward to multiple interfaces. Compare this to the actual default multicast address that is limited to specifying a single output interface for compatibility with existing functionality in other systems.

Another command sets the default multicast route:

```
void
Ipv4StaticRouting::SetDefaultMulticastRoute (uint32_t outputInterface);
```

This is the multicast equivalent of the unicast version SetDefaultRoute. We tell the routing system what to do in the case where a specific route to a destination multicast group is not found. The system forwards packets out the specified interface in the hope that "something out there" knows better how to route the packet. This method is only used in initially sending packets off of a host. The default multicast route is not consulted during forwarding – exact routes must be specified using AddMulticastRoute for that case.

Since we're basically sending packets to some entity we think may know better what to do, we don't pay attention to "subtleties" like origin address, nor do we worry about forwarding out multiple interfaces. If the default multicast route is set, it is returned as

the selected route from LookupStatic irrespective of origin or multicast group if another specific route is not found.

Finally, a number of additional functions are provided to fetch and remove multicast routes:

```
uint32_t GetNMulticastRoutes (void) const;

Ipv4MulticastRoute *GetMulticastRoute (uint32_t i) const;

Ipv4MulticastRoute *GetDefaultMulticastRoute (void) const;

bool RemoveMulticastRoute (Ipv4Address origin,
                           Ipv4Address group,
                           uint32_t inputInterface);

void RemoveMulticastRoute (uint32_t index);
```

11.7 Global centralized routing

Presently, global centralized IPv4 *unicast* routing over both point-to-point and shared (CSMA) links is supported. The global centralized routing will be modified in the future to reduce computations once profiling finds the performance bottlenecks.

11.8 Global Unicast Routing API

The public API is very minimal. User scripts include the following:

```
#include "ns3/global-route-manager.h"
```

After IP addresses are configured, the following function call will cause all of the nodes that have an Ipv4 interface to receive forwarding tables entered automatically by the GlobalRouteManager:

```
GlobalRouteManager::PopulateRoutingTables ();
```

Note: A reminder that the wifi NetDevice is not yet supported (only CSMA and PointToPoint).

It is possible to call this function again in the midst of a simulation using the following additional public function:

```
GlobalRouteManager::RecomputeRoutingTables ();
```

which flushes the old tables, queries the nodes for new interface information, and rebuilds the routes.

For instance, this scheduling call will cause the tables to be rebuilt at time 5 seconds:

```
Simulator::Schedule (Seconds (5),&GlobalRouteManager::RecomputeRoutingTables);
```

11.9 Global Routing Implementation

A singleton object (GlobalRouteManager) is responsible for populating the static routes on each node, using the public Ipv4 API of that node. It queries each node in the topology for a "globalRouter" interface. If found, it uses the API of that interface to obtain a "link

state advertisement (LSA)" for the router. Link State Advertisements are used in OSPF routing, and we follow their formatting.

The GlobalRouteManager populates a link state database with LSAs gathered from the entire topology. Then, for each router in the topology, the GlobalRouteManager executes the OSPF shortest path first (SPF) computation on the database, and populates the routing tables on each node.

The quagga (http://www.quagga.net) OSPF implementation was used as the basis for the routing computation logic. One benefit of following an existing OSPF SPF implementation is that OSPF already has defined link state advertisements for all common types of network links:

- point-to-point (serial links)
- point-to-multipoint (Frame Relay, ad hoc wireless)
- non-broadcast multiple access (ATM)
- broadcast (Ethernet)

Therefore, we think that enabling these other link types will be more straightforward now that the underlying OSPF SPF framework is in place.

Presently, we can handle IPv4 point-to-point, numbered links, as well as shared broadcast (CSMA) links, and we do not do equal-cost multipath.

The GlobalRouteManager first walks the list of nodes and aggregates a GlobalRouter interface to each one as follows:

```
typedef std::vector < Ptr<Node> >::iterator Iterator;
for (Iterator i = NodeList::Begin (); i != NodeList::End (); i++)
  {
    Ptr<Node> node = *i;
    Ptr<GlobalRouter> globalRouter = CreateObject<GlobalRouter> (node);
    node->AggregateObject (globalRouter);
  }
```

This interface is later queried and used to generate a Link State Advertisement for each router, and this link state database is fed into the OSPF shortest path computation logic. The Ipv4 API is finally used to populate the routes themselves.

11.10 Optimized Link State Routing (OLSR)

This is the first dynamic routing protocol for `ns-3`. The implementation is found in the src/routing/olsr directory, and an example script is in examples/simple-point-to-point-olsr.cc.

The following commands will enable OLSR in a simulation.

```
olsr::EnableAllNodes ();  // Start OLSR on all nodes
olsr::EnableNodes(InputIterator begin, InputIterator end); // Start on
  // a list of nodes
olsr::EnableNode (Ptr<Node> node);  // Start OLSR on "node" only
```

Once instantiated, the agent can be started with the Start() command, and the OLSR "main interface" can be set with the SetMainInterface() command. A number of protocol constants are defined in olsr-agent-impl.cc.

12 Wifi NetDevice

ns-3 nodes can contain a collection of NetDevice objects, much like an actual computer contains separate interface cards for Ethernet, Wifi, Bluetooth, etc. This chapter describes the ns-3 WifiNetDevice and related models. By adding WifiNetDevice objects to ns-3 nodes, one can create models of 802.11-based infrastructure and ad hoc networks.

12.1 Overview of the model

Note: This overview is taken largely from the Doxygen for the WifiNetDevice module.

The set of 802.11 models provided in ns-3 attempts to provide an accurate MAC-level implementation of the 802.11 specification and to provide a not-so-slow PHY-level model of the 802.11a specification.

The current implementation provides roughly four levels of models:

- the **PHY layer models**
- the so-called **MAC low models**: they implement DCF
- the so-called **MAC high models**: they implement the MAC-level beacon generation, probing, and association state machines, and
- a set of **Rate control algorithms** used by the MAC low models

There are presently three **MAC high models**:

1. a simple adhoc state machine that does not perform any kind of beacon generation, probing, or association. This state machine is implemented by the `ns3::AdhocWifiNetDevice` and `ns3::MacHighAdhoc` classes.
2. an active probing and association state machine that handles automatic re-association whenever too many beacons are missed is implemented by the `ns3::NqstaWifiNetDevice` and `ns3::MacHighNqsta` classes.
3. an access point that generates periodic beacons, and that accepts every attempt to associate. This AP state machine is implemented by the `ns3::NqapWifiNetDevice` and `ns3::MacHighNqap` classes.

The **MAC low layer** is split into three components:

1. `ns3::MacLow` which takes care of RTS/CTS/DATA/ACK transactions.
2. `ns3::DcfManager` and `ns3::DcfState` which implements the DCF function.
3. `ns3::DcaTxop` which handles the packet queue, packet fragmentation, and packet retransmissions if they are needed.

There are also several **rate control algorithms** that can be used by the Mac low layer:

- `ns3::ArfMacStations`
- `ns3::AArfMacStations`
- `ns3::IdealMacStations`
- `ns3::CrMacStations`
- `ns3::OnoeMacStations`
- `ns3::AmrrMacStations`

The PHY layer implements a single model in the `ns3::WifiPhy class`: the physical layer model implemented there is described fully in a paper entitled "Yet Another Network Simulator".

In ns-3, nodes can have multiple WifiNetDevices on separate channels, and the WifiNetDevice can coexist with other device types; this removes an architectural limitation found in ns-2. Presently, however, there is no model for cross-channel interference or coupling.

The source code for the Wifi NetDevice lives in the directory `src/devices/wifi`.

12.2 Using the WifiNetDevice

Users who use the low-level ns-3 API and who wish to add a WifiNetDevice to their node must create an instance of a WifiNetDevice, plus a number of consitutent objects, and bind them together appropriately (the WifiNetDevice is very modular in this regard, for future extensibility). At the low-level API, this can be done with about 20 lines of code (see `ns3::WifiHelper::Install` and `ns3::YansWifiPhyHelper::Create`). They also must create, at some point, a WifiChannel, which also contains a number of constituent objects (see `ns3::YansWifiChannelHelper::Create`).

However, a few helpers are available for users to add these devices and channels with only a few lines of code, if they are willing to use defaults, and the helpers provide additional API to allow the passing of attribute values to change default values. The scripts in `src/examples` can be browsed to see how this is done.

12.2.1 YansWifiChannelHelper

The YansWifiChannelHelper has an unusual name. Readers may wonder why it is named this way. The reference is to the yans simulator, from which this model is taken. The helper can be used to create a WifiChannel with a default PropagationLoss and PropagationDelay model. Specifically, the default is a channel model with a propagation delay equal to a constant, the speed of light, and a propagation loss based on a log distance model with a reference loss of 46.6777 dB at reference distance of 1m.

Users will typically type code such as:

```
YansWifiChannelHelper wifiChannelHelper = YansWifiChannelHelper::Default ();
Ptr<WifiChannel> wifiChannel = wifiChannelHelper.Create ();
```

to get the defaults. Note the distinction above in creating a helper object vs. an actual simulation object. In ns-3, helper objects (used at the helper API only) are created on the stack (they could also be created with operator new and later deleted). However, the actual ns-3 objects typically inherit from `class ns3::Object` and are assigned to a smart pointer. See the chapter on `Object model` for a discussion of the ns-3 object model, if you are not familiar with it.

Todo: Add notes about how to configure attributes with this helper API

12.2.2 YansWifiPhyHelper

Physical devices (base class `ns3::Phy`) connect to `ns3::Channel` models in ns-3. We need to create Phy objects appropriate for the YansWifiChannel; here the `YansWifiPhyHelper` will do the work.

The YansWifiPhyHelper class configures an object factory to create instances of a `YansWifiPhy` and adds some other objects to it, including possibly a supplemental ErrorRateModel and a pointer to a MobilityModel. The user code is typically:

```
YansWifiPhyHelper wifiPhyHelper = YansWifiPhyHelper::Default ();
wifiPhyHelper.SetChannel (wifiChannel);
```

Note that we haven't actually created any WifiPhy objects yet; we've just prepared the YansWifiPhyHelper by telling it which channel it is connected to. The phy objects are created in the next step.

12.2.3 WifiHelper

We're now ready to create WifiNetDevices. First, let's create a WifiHelper with default settings:

```
WifiHelper wifiHelper = WifiHelper::Default ();
```

What does this do? It sets the RemoteStationManager to `ns3::ArfWifiManager` and the upper MAC to `ns3::AdhocWifiMac` by default (which can be overridden by other arguments). Now, let's use the wifiPhyHelper created above to install WifiNetDevices on a set of nodes in a NodeContainer `"c"`:

```
NetDeviceContainer wifiContainer = WifiHelper::Install (wifiPhyHelper, c);
```

This creates the WifiNetDevice which includes also a WifiRemoteStationManager, a WifiMac, and a WifiPhy (connected to the matching WifiChannel).

There are many ns-3 `Attributes` that can be set on the above helpers to deviate from the default behavior; the example scripts show how to do some of this reconfiguration.

12.2.4 AdHoc WifiNetDevice configuration

This is a typical example of how a user might configure an adhoc network.

Write me

12.2.5 Infrastructure (Access Point and clients) WifiNetDevice configuration

This is a typical example of how a user might configure an access point and a set of clients.

Write me

12.3 The WifiChannel and WifiPhy models

The WifiChannel subclass can be used to connect together a set of `ns3::WifiNetDevice` network interfaces. The class `ns3::WifiPhy` is the object within the WifiNetDevice that receives bits from the channel. A WifiChannel contains a `ns3::PropagationLossModel` and a `ns3::PropagationDelayModel` which can be overridden by the WifiChannel::SetPropagationLossModel and the WifiChannel::SetPropagationDelayModel methods. By default, no propagation models are set.

The WifiPhy models an 802.11a channel, in terms of frequency, modulation, and bit rates, and interacts with the PropagationLossModel and PropagationDelayModel found in the channel.

This section summarizes the description of the BER calculations found in the yans paper taking into account the Forward Error Correction present in 802.11a and describes the

algorithm we implemented to decide whether or not a packet can be successfully received. See "Yet Another Network Simulator" for more details.

The PHY layer can be in one of three states:

1. TX: the PHY is currently transmitting a signal on behalf of its associated MAC
2. RX: the PHY is synchronized on a signal and is waiting until it has received its last bit to forward it to the MAC.
3. IDLE: the PHY is not in the TX or RX states.

When the first bit of a new packet is received while the PHY is not IDLE (that is, it is already synchronized on the reception of another earlier packet or it is sending data itself), the received packet is dropped. Otherwise, if the PHY is IDLE, we calculate the received energy of the first bit of this new signal and compare it against our Energy Detection threshold (as defined by the Clear Channel Assessment function mode 1). If the energy of the packet k is higher, then the PHY moves to RX state and schedules an event when the last bit of the packet is expected to be received. Otherwise, the PHY stays in IDLE state and drops the packet.

The energy of the received signal is assumed to be zero outside of the reception interval of packet k and is calculated from the transmission power with a path-loss propagation model in the reception interval. where the path loss exponent, n, is chosen equal to 3, the reference distance, d_0 is choosen equal to $1.0m$ and the reference energy is based based on a Friis propagation model.

When the last bit of the packet upon which the PHY is synchronized is received, we need to calculate the probability that the packet is received with any error to decide whether or not the packet on which we were synchronized could be successfully received or not: a random number is drawn from a uniform distribution and is compared against the probability of error.

To evaluate the probability of error, we start from the piecewise linear functions shown in Figure Figure 12.1 and calculate the SNIR function.

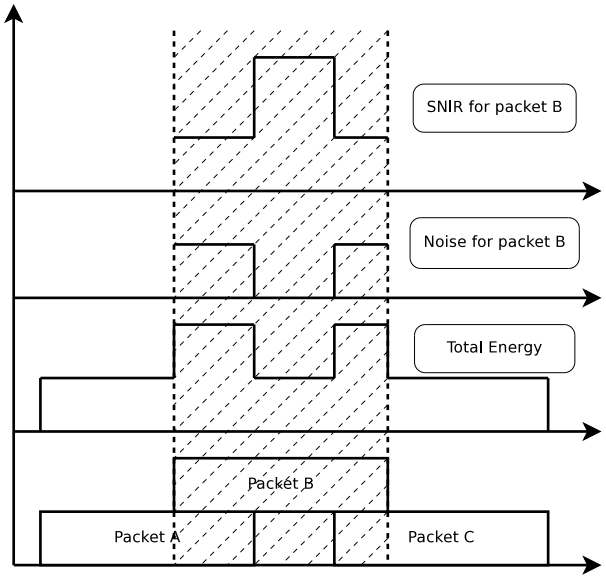


Figure 12.1: SNIR function over time

From the SNIR function we can derive bit error rates for BPSK and QAM modulations. Then, for each interval l where BER is constant, we define the upper bound of a probability that an error is present in the chunk of bits located in the interval l for packet k. If we assume an AWGN channel, binary convolutional coding (which is the case in 802.11a) and hard-decision Viterbi decoding, the error rate is thus derived, and the packet error probability for packet k can be computed..

12.3.1 WifiChannel configuration

WifiChannel models include both a PropagationDelayModel and a PropagationLossModel. The following PropagationDelayModels are available:

- ConstantSpeedPropagationDelayModel
- RandomPropagationDelayModel

The following PropagationLossModels are available:

- RandomPropagationLossModel
- FriisPropagationLossModel
- LogDistancePropagationLossModel
- JakesPropagationLossModel
- CompositePropagationLossModel

12.4 The MAC model

The 802.11 Distributed Coordination Function is used to calculate when to grant access to the transmission medium. While implementing the DCF would have been particularly easy if we had used a recurring timer that expired every slot, we chose to use the method

described in *(missing reference here from Yans paper)* where the backoff timer duration is lazily calculated whenever needed since it is claimed to have much better performance than the simpler recurring timer solution.

The higher-level MAC functions are implemented in a set of other C++ classes and deal with:

- packet fragmentation and defragmentation,
- use of the rts/cts protocol,
- rate control algorithm,
- connection and disconnection to and from an Access Point,
- the MAC transmission queue,
- beacon generation,
- etc.

12.5 Wifi Attributes

The WifiNetDevice makes heavy use of the ns-3 Chapter 3 [Attributes], page 12 subsystem for configuration and default value management. Presently, approximately 100 values are stored in this system.

For instance, class `ns-3::WifiMac` exports these attributes:

- CtsTimeout: When this timeout expires, the RTS/CTS handshake has failed.
- AckTimeout: When this timeout expires, the DATA/ACK handshake has failed.
- Sifs: The value of the SIFS constant.
- EifsNoDifs: The value of EIFS-DIFS
- Slot: The duration of a Slot.
- Pifs: The value of the PIFS constant.
- MaxPropagationDelay: The maximum propagation delay. Unused for now.
- MaxMsduSize: The maximum size of an MSDU accepted by the MAC layer.This value conforms to the specification.
- Ssid: The ssid we want to belong to.

12.6 Wifi Tracing

This needs revised/updating based on the latest Doxygen

ns-3 has a sophisticated tracing infrastructure that allows users to hook into existing trace sources, or to define and export new ones.

Wifi-related trace sources that are available by default include:

- `ns3::WifiNetDevice`
  - Rx: Received payload from the MAC layer.
  - Tx: Send payload to the MAC layer.
- `ns3::WifiPhy`
  - State: The WifiPhy state
  - RxOk: A packet has been received successfully.

- RxError: A packet has been received unsuccessfully.
- Tx: Packet transmission is starting.

Briefly, this means, for example, that a user can hook a processing function to the "State" tracing hook above and be notified whenever the WifiPhy model changes state.

13 CSMA NetDevice

This is the introduction to CSMA NetDevice chapter, to complement the Csma model doxygen.

13.1 Overview of the model

The ns-3 CSMA device models a simple bus network in the spirit of Ethernet. Although it does not model any real physical network you could ever build or buy, it does provide some very useful functionality.

Typically when one thinks of a bus network Ethernet or IEEE 802.3 comes to mind. Ethernet uses CSMA/CD (Carrier Sense Multiple Access with Collision Detection with exponentially increasing backoff to contend for the shared transmission medium. The ns-3 CSMA device models only a portion of this process, using the nature of the globally available channel to provide instantaneous (faster than light) carrier sense and priority-based collision "avoidance." Collisions in the sense of Ethernet never happen and so the ns-3 CSMA device does not model collision detection, nor will any transmission in progress be "jammed."

13.1.1 CSMA Layer Model

There are a number of conventions in use for describing layered communications architectures in the literature and in textbooks. The most common layering model is the ISO seven layer reference model. In this view the CsmaNetDevice and CsmaChannel pair occupies the lowest two layers – at the physical (layer one), and data link (layer two) positions. Another important reference model is that specified by RFC 1122, "Requirements for Internet Hosts – Communication Layers." In this view the CsmaNetDevice and CsmaChannel pair occupies the lowest layer – the link layer. There is also a seemingly endless litany of alternative descriptions found in textbooks and in the literature. We adopt the naming conventions used in the IEEE 802 standards which speak of LLC, MAC, MII and PHY layering. These acronyms are defined as:

- LLC: Logical Link Control;
- MAC: Media Access Control;
- MII: Media Independent Interface;
- PHY: Physical Layer.

In this case the *LLC* and *MAC* are sublayers of the OSI data link layer and the *MII* and *PHY* are sublayers of the OSI physical layer.

The "top" of the CSMA device defines the transition from the network layer to the data link layer. This transition is performed by higher layers by calling either CsmaNetDevice::Send or CsmaNetDevice::SendFrom.

In contrast to the IEEE 802.3 standards, there is no precisely specified PHY in the CSMA model in the sense of wire types, signals or pinouts. The "bottom" interface of the CsmaNetDevice can be thought of as as a kind of Media Independent Interface (MII) as seen in the "Fast Ethernet" (IEEE 802.3u) specifications. This MII interface fits into a corresponding media independent interface on the CsmaChannel. You will not find the equivalent of a 10BASE-T or a 1000BASE-LX PHY.

The CsmaNetDevice calls the CsmaChannel through a media independent interface. There is a method defined to tell the channel when to start "wiggling the wires" using the method CsmaChannel::TransmitStart, and a method to tell the channel when the transmission process is done and the channel should begin propagating the last bit across the "wire": CsmaChannel::TransmitEnd.

When the TransmitEnd method is executed, the channel will model a single uniform signal propagation delay in the medium and deliver copes of the packet to each of the devices attached to the packet via the CsmaNetDevice::Receive method.

There is a "pin" in the device media independent interface corresponding to "COL" (collision). The state of the channel may be sensed by calling CsmaChannel::GetState. Each device will look at this "pin" before starting a send and will perform appropriate backoff operations if required.

Properly received packets are forwarded up to higher levels from the CsmaNetDevice via a callback mechanism. The callback function is initialized by the higher layer (when the net device is attached) using CsmaNetDevice::SetReceiveCallback and is invoked upon "proper" reception of a packet by the net device in order to forward the packet up the protocol stack.

13.2 CSMA Channel Model

The class CsmaChannel models the actual transmission medium. There is no fixed limit for the number of devices connected to the channel. The CsmaChannel models a data rate and a speed-of-light delay which can be accessed via the attributes "DataRate" and "Delay" respectively. The data rate provided to the channel is used to set the data rates used by the transmitter sections of the CSMA devices connected to the channel. There is no way to independently set data rates in the devices. Since the data rate is only used to calculate a delay time, there is no limitation (other than by the data type holding the value) on the speed at which CSMA channels and devices can operate; and no restriction based on any kind of PHY characteristics.

The CsmaChannel has three states, `IDLE`, `TRANSMITTING` and `PROPAGATING`. These three states are "seen" instantaneously by all devices on the channel. By this we mean that if one device begins or ends a simulated transmission, all devices on the channel are *immediately* aware of the change in state. There is no time during which one device may see an `IDLE` channel while another device physically further away in the collision domain may have begun transmitting with the associated signals not propagated down the channel to other devices. Thus there is no need for collision detection in the CsmaChannel model and it is not implemented in any way.

We do, as the name indicates, have a Carrier Sense aspect to the model. Since the simulator is single threaded, access to the common channel will be serialized by the simulator. This provides a deterministic mechanism for contending for the channel. The channel is allocated (transitioned from state `IDLE` to state `TRANSMITTING`) on a first-come first-served basis. The channel always goes through a three state process:

```
IDLE -> TRANSMITTING -> PROPAGATING -> IDLE
```

The `TRANSMITTING` state models the time during which the source net device is actually wiggling the signals on the wire. The `PROPAGATING` state models the time after the last bit was sent, when the signal is propagating down the wire to the "far end."

The transition to the `TRANSMITTING` state is driven by a call to CsmaChannel::TransmitStart which is called by the net device that transmits the packet. It is the responsibility of that device to end the transmission with a call to CsmaChannel::TransmitEnd at the appropriate simulation time that reflects the time elapsed to put all of the packet bits on the wire. When TransmitEnd is called, the channel schedules an event corresponding to a single speed-of-light delay. This delay applies to all net devices on the channel identically. You can think of a symmetrical hub in which the packet bits propagate to a central location and then back out equal length cables to the other devices on the channel. The single "speed of light" delay then corresponds to the time it takes for: 1) a singal to propagate from one CsmaNetDevice through its cable to the hub; plus 2) the time it takes for the hub to forward the packet out a port; plus 3) the time it takes for the signal in question to propagate to the destination net device.

The CsmaChannel models a broadcast medium so the packet is delivered to all of the devices on the channel (including the source) at the end of the propagation time. It is the responsibility of the sending device to determine whether or not it receives a packet broadcast over the channel.

The CsmaChannel provides following Attributes:

- DataRate: The bitrate for packet transmission on connected devices;
- Delay: The speed of light transmission delay for the channel.

13.3 CSMA Net Device Model

The CSMA network device appears somewhat like an Ethernet device. The CsmaNetDevice provides following Attributes:

- Address: The Mac48Address of the device;
- SendEnable: Enable packet transmission if true;
- ReceiveEnable: Enable packet reception if true;
- EncapsulationMode: Type of link layer encapsulation to use;
- RxErrorModel: The receive error model;
- TxQueue: The trasmit queue used by the device;
- InterframeGap: The optional time to wait between "frames";
- Rx: A trace source for received packets;
- Drop: A trace source for dropped packets.

The CsmaNetDevice supports the assignment of a "receive error model." This is an ErrorModel object that is used to simulate data corruption on the link.

Packets sent over the CsmaNetDevice are always routed through the transmit queue to provide a trace hook for packets sent out over the network. This transmit queue can be set (via attribute) to model different queueing strategies.

Also configurable by attribute is the encapsulation method used by the device. Every packet gets an EthernetHeader that includes the destination and source MAC addresses, and a length/type field. Every packet also gets an EthernetTrailer which includes the FCS. Data in the packet may be encapsulated in different ways.

By default, or by setting the "EncapsulationMode" attribute to "Dix", the encapsulation is according to the DEC, Intel, Xerox standard. This is sometimes called EthernetII framing and is the familiar destination MAC, source MAC, EtherType, Data, CRC format.

If the "EncapsulationMode" attribute is set to "Llc", the encapsulation is by LLC SNAP. In this case, a SNAP header is added that contains the EtherType (IP or ARP).

The other implemented encapsulation modes are IP_ARP (set "EncapsulationMode" to "IpArp") in which the length type of the Ethernet header receives the protocol number of the packet; or ETHERNET_V1 (set "EncapsulationMode" to "EthernetV1") in which the length type of the Ethernet header receives the length of the packet. A "Raw" encapsulation mode is defined but not implemented – use of the RAW mode results in an assertion.

Note that all net devices on a channel must be set to the same encapsulation mode for correct results. The encapsulation mode is not sensed at the receiver.

The CsmaNetDevice implements a random exponential backoff algorithm that is executed if the channel is determined to be busy (`TRANSMITTING` or `PPROPAGATING`) when the device wants to start propagating. This results in a random delay of up to pow (2, retries) - 1 microseconds before a retry is attempted. The default maximum number of retries is 1000.

13.4 Using the CsmaNetDevice

The CSMA net devices and channels are typically created and configured using the associated `CsmaHelper` object. The various ns3 device dhelpers generatlly work in a simlar way, and their use is seen in many of our example programs.

The conceptual model of interest is that of a bare computer "husk" into which you plug net devices. The bare computers are created using a `NodeContainer` helper. You just ask this helper to create as many computers (we call them `Nodes`) as you need on your network:

```
NodeContainer csmaNodes;
csmaNodes.Create (nCsmaNodes);
```

Once you have your nodes, you need to instantiate a `CsmaHelper` and set any attributes you may want to change.

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

csma.SetDeviceAttribute ("EncapsulationMode", StringValue ("Dix"));
csma.SetDeviceAttribute ("FrameSize", UintegerValue (2000));
```

Once the attributes are set, all that remains is to create the devices and install them on the required nodes, and to connect the devices together using a CSMA channel. When we create the net devices, we add them to a container to allow you to use them in the future. This all takes just one line of code.

```
NetDeviceContainer csmaDevices = csma.Install (csmaNodes);
```

13.5 CSMA Tracing

Like all ns-3 devices, the CSMA Model provides a number of trace sources. These trace sources can be hooked using your own custom trace code, or you can use our helper functions to arrange for tracing to be enabled on devices you specify.

13.5.1 Upper-Level (MAC) Hooks

From the point of view of tracing in the net device, there are several interesting points to insert trace hooks. A convention inherited from other simulators is that packets destined for transmission onto attached networks pass through a single "transmit queue" in the net device. We provide trace hooks at this point in packet flow, which corresponds (abstractly) only to a transition from the network to data link layer, and call them collectively the device MAC hooks.

When a packet is sent to the CSMA net device for transmission it always passes through the transmit queue. The transmit queue in the CsmaNetDevice inherits from Queue, and therefore inherits three trace sources:

- An Enqueue operation source (see Queue::m_traceEnqueue);
- A Dequeue operation source (see Queue::m_traceDequeue);
- A Drop operation source (see Queue::m_traceDrop).

The upper-level (MAC) trace hooks for the CsmaNetDevice are, in fact, exactly these three trace sources on the single transmit queue of the device.

The m_traceEnqueue event is triggered when a packet is placed on the transmit queue. This happens at the time that CsmaNetDevice::Send or CsmaNetDevice::SendFrom is called by a higher layer to queue a packet for transmission.

The m_traceDequeue event is triggered when a packet is removed from the transmit queue. Dequeues from the transmit queue can happen in three situations: 1) If the underlying channel is idle when the CsmaNetDevice::Send or CsmaNetDevice::SendFrom is called, a packet is dequeued from the transmit queue and immediately transmitted; 2) If the underlying channel is idle, a packet may be dequeued and immediately transmitted in an internal TransmitCompleteEvent that functions much like a transmit complete interrupt service routine; or 3) from the random exponential backoff handler if a timeout is detected.

Case (3) implies that a packet is dequeued from the transmit queue if it is unable to be transmittted according to the backoff rules. It is important to understand that this will appear as a Dequeued packet and it is easy to incorrectly assume that the packet was transmitted since it passed through the transmit queue. In fact, a packet is actually dropped by the net device in this case. The reason for this behavior is due to the definition of the Queue Drop event. The m_traceDrop event is, by defintion, fired when a packet cannot be enqueued on the transmit queue becasue it is full. This event only fires if the queue is full and we do not overload this event to indicate that the CsmaChannel is "full."

13.5.2 Lower-Level (PHY) Hooks

Similar to the upper level trace hooks, there are trace hooks available at the lower levels of the net device. We call these the PHY hooks. These events fire from the device methods that talk directly to the CsmaChannel.

The trace source m_dropTrace is called to indicate a packet that is dropped by the device. This happens in two cases: First, if the receive side of the net device is not enabled (see CsmaNetDevice::m_receiveEnable and the associated attribute "ReceiveEnable").

The m_dropTrace is also used to indicate that a packet was discarded as corrupt if a receive error model is used (see CsmaNetDevice::m_receiveErrorModel and the associated attribute "ReceiveErrorModel").

The other low-level trace source fires on reception of an accepted packet (see CsmaNetDevice::m_rxTrace). A packet is accepted if it is destined for the broadcast address, a multicast address, or to the MAC address assigned to the net device.

14 PointToPoint NetDevice

This is the introduction to PointToPoint NetDevice chapter, to complement the PointToPoint model doxygen.

14.1 Overview of the model

The ns-3 point-to-point model is of a very simple point to point data link connecting exactly two PointToPointNetDevice devices over an PointToPointChannel. This can be viewed as equivalent to a full duplex RS-232 or RS-422 link with null modem and no handshaking.

Data is encapsulated in the Point-to-Point Protocol (PPP – RFC 1661), however the Link Control Protocol (LCP) and associated state machine is not implemented. The PPP link is assumed to be established and authenticated at all times.

Data is not framed, therefore Address and Control fields will not be found. Since the data is not framed, there is no need to provide Flag Sequence and Control Escape octets, nor is a Frame Check Sequence appended. All that is required to implement non-framed PPP is to prepend the PPP protocol number for IP Version 4 which is the sixteen-bit number 0x21 (see http://www.iana.org/assignments/ppp-numbers).

The PointToPointNetDevice provides following Attributes:

- Address: The ns3::Mac48Address of the device (if desired);
- DataRate: The data rate (ns3::DataRate) of the device;
- TxQueue: The trasmit queue (ns3::Queue) used by the device;
- InterframeGap: The optional ns3::Time to wait between "frames";
- Rx: A trace source for received packets;
- Drop: A trace source for dropped packets.

The PointToPointNetDevice models a transmitter section that puts bits on a corresponding channel "wire." ‘The DataRate attribute specifies the number of bits per second that the device will simulate sending over the channel. In reality no bits are sent, but an event is scheduled for an elapsed time consistent with the number of bits in each packet and the specified DataRate. The implication here is that the receiving device models a receiver section that can receive any any data rate. Therefore there is no need, nor way to set a receive data rate in this model. By setting the DataRate on the transmitter of both devices connected to a given PointToPointChannel one can model a symmetric channel; or by setting different DataRates one can model an asymmetric channel (e.g., ADSL).

The PointToPointNetDevice supports the assignment of a "receive error model." This is an ErrorModel object that is used to simulate data corruption on the link.

14.2 Point-to-Point Channel Model

The point to point net devices are connected via an PointToPointChannel. This channel models two wires transmitting bits at the data rate specified by the source net device. There is no overhead beyond the eight bits per byte of the packet sent. That is, we do not model Flag Sequences, Frame Check Sequences nor do we "escape" any data.

The PointToPointChannel provides following Attributes:

- Delay: An ns3::Time specifying the speed of light transmission delay for the channel.

14.3 Using the PointToPointNetDevice

The PointToPoint net devices and channels are typically created and configured using the associated `PointToPointHelper` object. The various ns3 device helpers generally work in a simlar way, and their use is seen in many of our example programs and is also covered in the ns-3 tutorial.

The conceptual model of interest is that of a bare computer "husk" into which you plug net devices. The bare computers are created using a `NodeContainer` helper. You just ask this helper to create as many computers (we call them `Nodes`) as you need on your network:

```
NodeContainer nodes;
nodes.Create (2);
```

Once you have your nodes, you need to instantiate a `PointToPointHelper` and set any attributes you may want to change. Note that since this is a point-to-point (as compared to a point-to-mulipoint) there may only be two nodes with associated net devices connected by a PointToPointChannel.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

Once the attributes are set, all that remains is to create the devices and install them on the required nodes, and to connect the devices together using a PointToPoint channel. When we create the net devices, we add them to a container to allow you to use them in the future. This all takes just one line of code.

```
NetDeviceContainer devices = pointToPoint.Install (nodes);
```

14.4 PointToPoint Tracing

Like all ns-3 devices, the Point-to-Point Model provides a number of trace sources. These trace sources can be hooked using your own custom trace code, or you can use our helper functions to arrange for tracing to be enabled on devices you specify.

14.4.1 Upper-Level (MAC) Hooks

From the point of view of tracing in the net device, there are several interesting points to insert trace hooks. A convention inherited from other simulators is that packets destined for transmission onto attached networks pass through a single "transmit queue" in the net device. We provide trace hooks at this point in packet flow, which corresponds (abstractly) only to a transition from the network to data link layer, and call them collectively the device MAC hooks.

When a packet is sent to the Point-to-Point net device for transmission it always passes through the transmit queue. The transmit queue in the PointToPointNetDevice inherits from Queue, and therefore inherits three trace sources:

- An Enqueue operation source (see ns3::Queue::m_traceEnqueue);
- A Dequeue operation source (see ns3::Queue::m_traceDequeue);
- A Drop operation source (see ns3::Queue::m_traceDrop).

The upper-level (MAC) trace hooks for the PointToPointNetDevice are, in fact, exactly these three trace sources on the single transmit queue of the device.

The m_traceEnqueue event is triggered when a packet is placed on the transmit queue. This happens at the time that ns3::PointtoPointNetDevice::Send or ns3::PointToPointNetDevice::SendFrom is called by a higher layer to queue a packet for transmission. An Enqueue trace event firing should be interpreted as only indicating that a higher level protocol has sent a packet to the device.

The m_traceDequeue event is triggered when a packet is removed from the transmit queue. Dequeues from the transmit queue can happen in two situations: 1) If the underlying channel is idle when PointToPointNetDevice::Send is called, a packet is dequeued from the transmit queue and immediately transmitted; 2) a packet may be dequeued and immediately transmitted in an internal TransmitCompleteEvent that functions much like a transmit complete interrupt service routine. An Dequeue trace event firing may be viewed as indicating that the PointToPointNetDevice has begun transmitting a packet.

14.4.2 Lower-Level (PHY) Hooks

Similar to the upper level trace hooks, there are trace hooks available at the lower levels of the net device. We call these the PHY hooks. These events fire from the device methods that talk directly to the PointToPointChannel.

The trace source m_dropTrace is called to indicate a packet that is dropped by the device. This happens when a packet is discarded as corrupt due to a receive error model indication (see ns3::ErrorModel and the associated attribute "ReceiveErrorModel").

The other low-level trace source fires on reception of a packet (see ns3::PointToPointNetDevice::m_rxTrace) from the PointToPointChannel.

15 Troubleshooting

This chapter posts some information about possibly common errors in building or running ns-3 programs.

Please note that the wiki (http://www.nsnam.org/wiki/index.php/Troubleshooting) may have contributed items.

15.1 Build errors

15.2 Run-time errors

Sometimes, errors can occur with a program after a successful build. These are run-time errors, and can commonly occur when memory is corrupted or pointer values are unexpectedly null.

Here is an example of what might occur:

```
ns-old:~/ns-3-nsc$ ./waf --run tcp-point-to-point
Entering directory `/home/tomh/ns-3-nsc/build'
Compilation finished successfully
Command ['/home/tomh/ns-3-nsc/build/debug/examples/tcp-point-to-point'] exited with code -1
```

The error message says that the program terminated unsuccessfully, but it is not clear from this information what might be wrong. To examine more closely, try running it under the gdb debugger:

```
ns-old:~/ns-3-nsc$ ./waf --run tcp-point-to-point --command-template="gdb %s"
Entering directory `/home/tomh/ns-3-nsc/build'
Compilation finished successfully
GNU gdb Red Hat Linux (6.3.0.0-1.134.fc5rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db library "/lib/

(gdb) run
Starting program: /home/tomh/ns-3-nsc/build/debug/examples/tcp-point-to-point
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xf5c000

Program received signal SIGSEGV, Segmentation fault.
0x0804aa12 in main (argc=1, argv=0xbfdfefa4)
    at ../examples/tcp-point-to-point.cc:136
136       Ptr<Socket> localSocket = socketFactory->CreateSocket ();
(gdb) p localSocket
$1 = {m_ptr = 0x3c5d65}
(gdb) p socketFactory
$2 = {m_ptr = 0x0}
```

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

Note first the way the program was invoked– pass the command to run as an argument to the command template "gdb %s".

This tells us that there was an attempt to dereference a null pointer socketFactory.

Let's look around line 136 of tcp-point-to-point, as gdb suggests:

```
Ptr<SocketFactory> socketFactory = n2->GetObject<SocketFactory> (Tcp::iid);
Ptr<Socket> localSocket = socketFactory->CreateSocket ();
localSocket->Bind ();
```

The culprit here is that the return value of GetObject is not being checked and may be null.

Sometimes you may need to use the valgrind memory checker for more subtle errors. Again, you invoke the use of valgrind similarly:

```
ns-old:~/ns-3-nsc$ ./waf --run tcp-point-to-point --command-template="valgrind %s"
```

(Index is nonexistent)